

EXPLORE GO CRYPTOGRAPHY

“I laughed, I learned. It's a great book.”

—Dian Amencourt



A GOPHER'S GUIDE TO CIPHERS

JOHN ARUNDEL

Explore Go: Cryptography

John Arundel

Bitfield Consulting

February 12, 2025

© 2025 John Arundel

Explore Go: Cryptography

[Praise for *Explore Go: Cryptography*](#)

[Introduction](#)

[Cryptography](#)

[Why learn about cryptography?](#)

[What do you need to know?](#)

[About the book](#)

[What you'll learn](#)

[How to use this book](#)

[1. Ciphers](#)

[Codes and ciphers](#)

[A shift cipher](#)

[Key points](#)

[Wheels within wheels](#)

[Enciphering](#)

[One byte at a time](#)

[Assume a spherical cow](#)

[From behaviour to test](#)

[Choose your own adventure](#)

[A first test](#)

[Anatomy of a test](#)

[When should the test fail?](#)

[The world's greatest bug detector](#)

[A function about nothing](#)

[Cutting code](#)

[The smarty-pants answer](#)

[We're gonna need a bigger test](#)

[A table of cases](#)

[Combining cases](#)

[A suspiciously similar test](#)

[A test case struct](#)

[Looping over the cases](#)

[Lighting a fire under the test](#)

[Introducing subtests](#)

[Subtests and t.Run](#)
[A litany of failure](#)
[Adding more cases](#)
[A working table test for Encipher](#)

[2. Enciphering](#)

[A simple enciphering filter](#)
[A command package](#)
[Garbage in, garbage out](#)
[All about Eve](#)
[Reverse engineering](#)
[Variable keys](#)
[Adding a keyhole](#)
[New keys, new cases](#)
[Compiler complaints](#)
[Stochastic debugging](#)
[Getting back to green](#)
[Getting the key](#)
[Defining a key flag](#)
[Getting the flag value](#)
[Enciphering with arbitrary keys](#)

[3. Deciphering](#)

[Brute force and ignorance](#)
[A statistical vulnerability](#)
[How many possible keys are there?](#)
[How safe is your safe?](#)
[Doing it for the crack](#)
[Cribs](#)
[Designing a cracking function](#)
[First, we need Decipher](#)
[Do we even need a test, then?](#)
[New test, same old table](#)
[Look before you loop](#)
[A deciphering tool](#)
[Let's not complicate encipher](#)

[4. Cracking](#)

[Testing a crack function](#)
[Failure is an option](#)

[A key-guessing function](#)
[A command-line cracker](#)
[What do users want?](#)
[Crib constraints](#)
[Cracking a real ciphertext](#)
[Waiting for the tiger](#)
[The devil in the details](#)
[0x89 is the magic number](#)
[Printing the unprintable](#)

5. Keys

[Lost in keyspace](#)
[The space of all possible keys](#)
[How long should the key be?](#)
[A keyspace the size of the universe](#)
[Multi-byte keys](#)
[Embiggening the key](#)
[Updating the test cases](#)
[Bytes in, bytes out](#)
[Fixing Encipher](#)
[A magical solution](#)
[Constraining the key index](#)
[Modulate your enthusiasm](#)
[The modulus operator](#)
[Refactoring will continue until morale improves](#)

[Testing longer keys](#)
[Designing the test cases](#)
[Sharpening the tools](#)
[When the right answer is wrong](#)
[What kind of flag do we need?](#)

[The joy of hex\(adecimal\)](#)
[Binary notation](#)
[A string flag](#)
[Where's the BEEF?](#)
[Changing frequencies](#)

6. Cribs

[Cracking long keys](#)
[What needs to change?](#)

[A byte at the problem](#)
[Length limitations](#)
[Checking our guesses](#)
[Knowing when to stop](#)
[A proper little cracker](#)
[Updating the crack tool](#)
[Applying brute force](#)
[Crib considerations](#)
[A false deciphering](#)
[On being the right size](#)

[7. Passwords](#)

[Security](#)
[Password spaces](#)
[Dictionary words](#)
[Extremely guessable passwords](#)
[Passphrases](#)

[Rules](#)

[Character sets](#)
[Password policies](#)
[You're really not helping](#)

[Maximising the keyspace](#)

[Unicode](#)

[Generation](#)

[What does “random” mean?](#)
[Predictability](#)
[Binary choices](#)
[Information](#)
[Knowledge](#)

[8. Blocks](#)

[Block ciphers](#)
[Why blocks?](#)
[The cipher.Block interface](#)
[Plugs and sockets](#)
[Adapting the shift cipher](#)
[The Encrypt method](#)
[Where's the key?](#)
[Fixing the key size](#)

A cipher constructor

The NewCipher function

If the key fits

A polite refusal

A special error value

Wrapping errors

Writing NewCipher

Implementing the cipher.Block interface

Testing Encrypt

Circumstances alter cases

Creating the cipher object

A test for Encrypt

Writing Encrypt and decrypt

Forgotten something?

9. Modes

Handling data in blocks

Updating the encipher tool

Block by block

The cipher.BlockMode interface

A simple block mode

Testing CryptBlocks

Creating the encrypter object

Implementing CryptBlocks

Using a BlockMode in encipher

Block alignment

When misaligned data attacks

Checking our assumptions

The long and short of it

A test about nothing

A panic-proof CryptBlocks

10. Padding

A padding scheme

Making ends meet

The “illegal pixel” problem

An unambiguous padding scheme

Testing Pad

Implementing Pad

[Writing Unpad](#)

[Padding input to encipher](#)

[Adding the padding](#)

[Checking the results](#)

[Deciphering](#)

[Adding the unpadding](#)

[A decrypter that implements BlockMode](#)

[Waiting for the tiger](#)

[11. Enumeration](#)

[Adventures in keyspace](#)

[A new test for cracking long keys](#)

[A block is a black box](#)

[Enumerating long keys](#)

[Designing a Next function](#)

[Testing Next](#)

[Big endians and little endians](#)

[A simple implementation](#)

[Checking our key guesses](#)

[The soul of a new cracker](#)

[We apologise for the delay](#)

[Benchmarking key cracking](#)

[Pick an easier problem](#)

[Understanding benchmark output](#)

[Cranking up the difficulty](#)

[Ballparking a realistic crack time](#)

[A test we might live to see pass](#)

[Updating the crack tool](#)

[Putting it all together](#)

[Will it crack?](#)

[Parallelisation](#)

[Big computers are too slow](#)

[Dense computers become black holes](#)

[Parallel cracking is still useful](#)

[But we won't implement it here](#)

[Strong keys can't be cracked](#)

[12. Entropy](#)

[Information](#)

Entropy: a measure of our ignorance

Entropy of digital messages

Compression

Enciphered data

Identifying ciphertexts

Complexity

Describing sequences

A generating algorithm

The shortest program that describes a sequence

Kolmogorov complexity

Complexity from simplicity

Security

Randomness is high entropy

Key generation

Pragmatic non-determinism

A little entropy goes a long way

Being unpredictable

13. Randomness

Pseudo-random generation

Randomness in games

The Fibonacci sequence

A simple random generator

A Fibonacci generator in Go

Repeatability

Seeding the RNG

Security problems

Periodicity

Distribution

Uniformity

“Secure enough” random generators

Environmental noise

The system entropy pool

Security is relative

You don’t need to outrun the bear

Keeping secrets from the gods

Hardware entropy sources

Defeating “God Emperor Eve”

[Quantum measurements](#)

[A quantum randomness source](#)

[Generating quantum keys](#)

[14. Chains](#)

[Visualising the problem](#)

[A completely random image](#)

[Separating metadata and pixel data](#)

[Hidden figures](#)

[Muddying the waters](#)

[What's wrong with this picture?](#)

[Mallory in the middle](#)

[ECB: a block-headed operating mode](#)

[Introducing Mallory](#)

[Block replay](#)

[Dropping and modifying blocks](#)

[More sophisticated modes](#)

[Counter mode \(CTR\)](#)

[Nonces](#)

[Cipher Block Chain \(CBC\)](#)

[Initialization Vector \(IV\)](#)

[Generating a secure IV](#)

[Implementing CBC mode](#)

[Enciphering in CBC mode](#)

[The updated encipher program](#)

[Updating the decipher program](#)

[The devil in disguise](#)

[15. Hashing](#)

[Message integrity](#)

[The night is dark, and full of errors](#)

[Bit-flipping and block-dropping](#)

[Integrity checking](#)

[Digests and hashing](#)

[Hash tables](#)

[Buckets and distribution](#)

[Collisions](#)

[Cryptographic hashing](#)

[Simple hash functions](#)

[A naïve algorithm](#)

[Implementing LenHash](#)

[Problems with LenHash](#)

[Preimage attacks](#)

[A slight improvement](#)

[Implementing SumHash](#)

[Testing SumHash](#)

[A preimage attack on SumHash](#)

[The avalanche effect](#)

[Real hash algorithms](#)

[MD5](#)

[SHA-1](#)

[SHA-256](#)

[Password hashing](#)

[Zero-knowledge secret storage](#)

[Password hashing requirements](#)

[Rainbow tables](#)

[Salting](#)

[Slow down, you move too fast](#)

[16. Coins](#)

[Cryptocurrency](#)

[Let there be “Bobcoin”](#)

[The problem of trust](#)

[A distributed digital ledger](#)

[Security](#)

[The double-spending problem](#)

[Ordering transactions](#)

[The blockchain](#)

[Doomed stubs](#)

[Integrity](#)

[Consensus](#)

[Proof of work](#)

[Up in smoke](#)

[Proof of stake](#)

[17. Authentication](#)

[Message integrity](#)

[Hash preimage attacks](#)

[Chosen ciphertext attacks](#)

[MAC](#)

[Length extension attacks](#)

[HMAC](#)

[Key exchange](#)

[The problem](#)

[Key splitting](#)

[Asymmetric encryption](#)

[Public and private keys](#)

[The Diffie-Hellman-Merkle protocol](#)

[A one-way function for key exchange](#)

[A DHM worked example](#)

[The reveal](#)

[Public-key cryptography](#)

[RSA](#)

[Signing](#)

[Authentication](#)

[Verification](#)

[The chain of trust](#)

[18. Cryptography](#)

[A little history](#)

[The origins of DES](#)

[Tripling down on DES](#)

[AES](#)

[Rijndael](#)

[The starting grid](#)

[Round and round](#)

[Confusion and diffusion](#)

[Putting it all together](#)

[Implementing AES](#)

[Getting ready to rumble](#)

[Ding ding, round one](#)

[The diffusion loop](#)

[The last round](#)

[AES encryption in practice](#)

[Enciphering with AES-CBC](#)

[Updating encipher and decipher](#)

[Encryption plus authentication](#)

[Enciphering with AES-GCM](#)

[The final final version](#)

[Weaknesses](#)

[Implementation fails](#)

[Mashing the keys](#)

[Tomorrow](#)

[The future is quantum](#)

[Quantum parallelism](#)

[The catch](#)

[Why your computer isn't quantum... yet](#)

[Post-quantum cryptography](#)

[Afterword](#)

[About this book](#)

[Who wrote this?](#)

[Feedback](#)

[Free updates to future editions](#)

[Join my Go Club](#)

[For the Love of Go](#)

[The Power of Go: Tools](#)

[Further reading](#)

[Credits](#)

[Acknowledgements](#)

Praise for *Explore Go: Cryptography*

I laughed, I learned. It's a great book that manages to explain complicated ideas in really simple, straightforward language.

—Dian Amencourt

One of the most accessible and even fun books about cryptography I've ever read. John writes vividly, with many a neat turn of phrase, and the text is full of wit and humour.

—Kurtis Connor

This is a remarkable book, practically fizzing with enthusiasm for its subject. It whisks you through the worlds of cryptography, mathematics, physics, and computer science, and shows you that they're all connected in surprising and thought-provoking ways. Highly enjoyable.

—Cuong Quoc

As a software engineer who works on a security product (I won't say which one), I've always felt like a bit of an imposter when it comes to cryptography: I hoped no one would ever ask me how a cipher actually works. After reading this book, I now feel a lot more confident. Thanks, John!

—Yuliana M

So friendly and easy to read. This has really unlocked a few things for me. I can't recommend this book highly enough!

—Joseph Adewale

John has a knack of writing almost luminously clear explanations that stick in your mind long after finishing the book.

—Patrick Devlin

Introduction

This book is all about keeping secrets. It shows you how to set up secret meeting places and a secret post office, and how to disguise your messages and maps. It shows you lots of secret codes and signals.

—Falcon Travis and Judy Hindley, [The KnowHow Book of Spycraft](#), 1975



First, why should you even read a book about cryptography? It's a valid question.

Cryptography

When I was a kid, one of my favourite books (and there were many) was [The KnowHow Book of Spycraft](#), which, as you might have gathered from the quotation above, is all about secret messages.

Kids love secret messages and spies, of course, along with dinosaurs, trains, and poop. But our obsessive interest in these subjects tends to fade as we grow older (except for dinosaurs, perhaps). So why, assuming you didn't

grow up to become a spy yourself, is it worth your while to read *this* book, which is also all about secret messages?

Why learn about cryptography?

Well, firstly, the subject is cool and interesting. I thought so when I was seven years old, evidently, and I still think so now, when I'm just a little older. It's okay to study and learn about things just because they're interesting, even if they don't really have any practical value. That's something kids don't need telling, but many adults have forgotten, sadly enough.

Actually, though, cryptography *is* rather important, and we rely on it every day of our lives, even though we might not be aware of it. The credit card or payment service you used to buy this book uses cryptographic algorithms and protocols to keep your transactions secure (we hope). The messages, photos, and videos that you exchange with friends, family, and co-workers are encrypted so that they can't fall into the wrong hands (we *better* hope). And so on.

If you use cryptocurrencies such as Bitcoin, well, guess what? It's not called "crypto" for nothing. And it's not just criminals and fraudsters who value the security and anonymity that comes from strong, modern cryptography: we all do. Your computer, your phone, your car—heck, probably even your fridge—contain hardware and software dedicated to cryptography. The internet relies on it. The banking system depends on it.

Basically, modern life is more or less built on cryptography. Isn't it a bit weird that most of us don't really know the first thing about it? Maybe it's time we did.

What do you need to know?

Let's start with what you *don't* need to know. You don't need to be able to design your own ciphers or cryptographic protocols. In fact, you shouldn't do that at all unless you're a real expert, and probably not even then.

Some of the smartest people who ever lived have designed cipher systems that turned out later to be insecure, in many cases woefully so. Even if you’re one of those people (and you did buy this book, after all), you’re much better off just using one of the modern cryptographic schemes in which no vulnerabilities have yet been found.

Even using a genuinely secure cipher, though, it’s still possible that you (or I) might make mistakes when incorporating it into software (indeed, it’s highly likely: this happens all the time). It’s incredibly easy to make small errors or omissions that render the resulting program dangerously insecure for its users: even worse, you (and they) might never know until it’s too late.

If you’re wise, then, you won’t try to write your own cryptography software either. Just use existing programs that have been written and debugged by lots of smart people and tested by many more.

To *use* cryptography securely, though, even something as simple as a password, you do need to know something about what *makes* things secure, or not. We can probably recognise a weak password when we see one (especially if it’s “password”), but what makes a strong password strong, and why? Similarly, we’re aware that some ciphers are weak and others are strong, but why?

The best way to answer that question, ironically enough, is to write your own cryptography software, and that’s what we’re going to do together in this book. I want to emphasise, though, that we’ll be doing this purely for fun and learning purposes. You shouldn’t use any of the software we build here to protect *actual* secrets (at least, not until we get to the last chapter).

Modern cryptography is extremely sophisticated, and if I tried to just explain that stuff right away it would probably kill both of us. Instead, we’ll work up to it, starting with something very simple, and gradually incorporating some of the many improvements in ciphers over the last two thousand years or so.

In the beginning, we’ll sweat the details and look at every nut and bolt, while it’s still easy to do so. As things get more complicated and closer to

the modern state of the art, there'll be a bit less detail and a bit more hand-waving, but that's okay. You'll still learn everything you want to know (and, indeed, *should* know), as a software engineer who occasionally has to dip a toe into cryptographic waters.

About the book

This book will explain the fundamental principles of ciphers, by introducing you to the simplest cipher scheme imaginable. You'll learn how it works, implement it in Go, and then you'll find out why it won't protect your secrets from anybody who really wants to know them (except perhaps your kid sister).

What you'll learn

You'll learn about cipher keys: how they work, what makes them weak or strong, how to generate them, and how to defeat them. You'll learn how to choose good passwords, and how to store and retrieve them securely, and you'll learn the “10 Do’s and 500 Don’ts” of writing software applications that handle passwords.

In the book, we'll gradually improve the security and usability of our toy cipher scheme, adding better keys, blocks, operating modes, hashing, authentication, and so on. We'll talk about predictability and randomness, message digests, cryptocurrency, and all sorts of other stuff that's fun and interesting (at least, if you're the kind of person who likes secret codes and dinosaurs).

There'll necessarily be a *little* bit of math. Now, computer programmers are often somewhat mathematically inclined, but many aren't, and that's okay. If you're one of the latter, don't worry. We will come across a few equations and formulas, but if they just look like illegible squiggles to you, it doesn't matter. You will certainly understand the general *principles* involved, and that's just fine.

(If you *are* a math whiz, you'll probably realise pretty soon that I am not. Please email me with a list of all the mistakes you find in the book: I always

enjoy hearing from readers.)

How to use this book

Throughout this book we'll be working together to develop some cryptographic tools and packages in Go. There are dozens of challenges for you to solve throughout the book, each designed to help you test your understanding of the concepts you've just learned.

If you get stuck on any of these, or just want to check your results, each challenge is accompanied by hints, and a complete sample solution, including tests. These are linked in the text, and you can also find them on GitHub here:

- <https://github.com/bitfield/eg-crypto>

You'll need to install Go on your computer, if you don't have it already. Follow the instructions on the Go website to download and install Go:

- <https://go.dev/learn/>

With that out of the way, let's begin!

1. Ciphers

If you reveal your secrets to the wind you should not blame the wind for revealing them to the trees.

—Kahlil Gibran, [“Sand and Foam”](#)



(photo by [Earl Grey Ltd](#))

Let's start with a quick introduction to the basics of *cryptography* (from the Greek for “secret writing”). First, let's distinguish *codes* from *ciphers*. These words are often used interchangeably, but strictly speaking they refer to two different things.

Codes and ciphers

To *encrypt* something is to hide or obscure its contents, and there are many ways to do this. One way is to use a *code*: an alternate, usually shorter, way of representing the text of a message.

For example, you've probably heard of the Morse code, in which each letter of the alphabet is represented by a distinct pattern of short and long sounds.

If you can transmit sound, though, why not just send a spoken message instead of a bunch of beeps?

The answer is that human speech is complex and covers a broad spectrum of frequencies, meaning that it takes up a lot of *bandwidth* (or, equivalently, that it needs a high *bitrate*). If you've ever struggled to understand someone or to make yourself understood over a poor-quality telephone connection, you'll know what I mean.

When there isn't much bandwidth available, for example in a noisy or lossy channel such as a long telegraph wire, or a low-power radio transmission, Morse and other codes are a far more efficient medium than speech for getting your message through, quickly and reliably.

Note that a code isn't necessarily intended to *hide* the contents of the message; it's often just a way of shortening it or making it easier to transmit reliably, as with the Morse code.

Codes *can* be used to send secret messages, though. For example, if we want to have a conversation that may be overheard, we might agree beforehand that the word "tomato" means "enemy in sight", and that "parmesan" means "activate the hidden sleeping-gas capsule". Then I can send you the following message *in clear* (that is, without needing to hide its contents):

TOMATO. PARMESAN.

If the message should be intercepted, the enemy won't be able to make anything of it. They'll probably just assume it's my shopping list. Note that it doesn't matter *what* code words we choose, only that we both agree on what they signify.

One drawback of such codes is that the list of code words and their meanings—the *code book*—tends to be large, and that makes it difficult to keep it secure. Should the code book fall into enemy hands, the code becomes useless until you can arrange to distribute a new code book to all your agents, which is expensive and time-consuming.

So another kind of *encryption*—message hiding—is to use a *cipher*. This works by substituting, usually, one *letter* for another, instead of a word or phrase. If the text is represented digitally (that is, by using numbers to represent letters), then the cipher operates by transforming one number into another, but the principle is the same.

A shift cipher

As long as both the sender and receiver (traditionally known as “Alice” and “Bob”, since that’s slightly more fun than just calling them A and B) agree on the cipher scheme, or *algorithm*, they can decrypt each other’s messages.

Let’s see an example. Suppose Alice and Bob agree that every time the letter *A* occurs in the *plaintext*—the original message—Alice will replace it with *B*. Each *B* in the plaintext will be replaced by *C*, and so on. In effect, this scheme just “adds one” to each letter to encipher it.

What does a plaintext like *HEY BOB U UP* look like, when enciphered using this scheme? If *H* becomes *I*, then the first letter of the *ciphertext* must be *I*. The second letter, *E*, becomes *F*, and so on. Encipher the rest of the message for yourself.

How can we *decipher* such a message, then? Simple: just reverse the process. For example, suppose Bob sends the following reply:

TVSF CBF DPNF PO PWFS

Can you work out whether Alice and Bob will be getting together tonight? Of course you can. Just “subtract one” from each letter to recover the plaintext. For example, *T* becomes *S*, *V* becomes *U*, and so on.

Key points

You don’t have to always shift the letters by *one* place; you can choose a different number if you like. Of course, the person reading the message will need to know this number too, but you can agree that between you in advance. Naturally, it needs to be kept secret: anyone who knows this

prearranged value will be able to decipher messages (and encipher them too).

This *key* is an important input to the cipher, along with the plaintext. Indeed, we can think of a cipher as *combining* the plaintext with the key, in some arbitrary way, to produce the ciphertext. Deciphering is the reverse of this process.

Wheels within wheels

A popular children's toy illustrates this idea: the “decoder ring” or *cipher wheel*, constructed from two rotating discs with the alphabet printed around them. It may not exactly be military-grade encryption, but it's an easy and fun way of exchanging secret messages with friends. Just choose your key (for example, 1) and rotate the outer disc by that number of places, and you're ready for enciphering and deciphering.

This kind of scheme—where we shift letters around the alphabet by some agreed number of places—is called a *shift* cipher, or sometimes a “Caesar” cipher, since Julius Caesar is popularly supposed to have used one to keep his messages secret. Not that it worked out awfully well for him in the end, but never mind.

Modern ciphers are a bit more complicated in their detailed workings than Caesar's shift cipher, but surprisingly similar in principle. So let's start with this one and work our way up.

Enciphering

We'll begin by trying to implement the shift cipher in a Go program. Our program should be able to take a message, encipher it using the shift cipher, and print out the resulting ciphertext. And let's generalise the idea of a “message” to any arbitrary data. Maybe Julius Caesar didn't have to worry about communicating things like graphics, video, or even binary executables, but we do.

One byte at a time

In general, the most flexible way to represent arbitrary data in Go is as a `[]byte`, that is, a slice of values each of which corresponds to one byte of data. A *slice* is the Go name for an ordered collection of values. You can think of the type `[]byte` as meaning “a bunch of bytes”.

For example, if you open and read a file from disk, or read from a network connection, you’ll find that you get the data as a bunch of bytes—a `[]byte`—regardless of what kind of data the bytes actually represent.

So the API to our shift cipher package (let’s call it `shift`, to be straightforward) is starting to take shape. We need to be able to accept data as bytes, transform it using the cipher scheme, and produce the resulting data as output, again in the form of bytes.

Even with our fairly simple cipher scheme, this still sounds like a hard problem. I’ll let you into one of the software engineering fraternity’s best-kept secrets now (make sure no one’s reading over your shoulder):

We don’t like to solve hard problems.

It *can* be done, of course, but it’s very much a last resort.

Instead, we much prefer to turn a complicated problem into a simple problem, and solve *that* first. Then it’s just a case of tweaking our solution until it works for the original problem, as well as our simplified version.

We’re not the only ones who like to do this; scientists call it the “[spherical cow](#)” technique, after the old joke about the farmer who made the mistake of asking a physicist for advice on improving milk production. “First,” said the physicist, after a little thought, “assume a spherical cow...”

Assume a spherical cow

The simplest, most “spherical” version of our problem that we can think of is just some function that transforms a slice of bytes by enciphering it with the key of 1. Let’s call this imaginary function `Encipher`.

For the moment, we won't worry about using different keys, and we'll also punt on the issue of getting the data in and out of the program. Let's get the spherical cow working first, then we'll add things step by step to make it more realistic.

Before trying to implement any function, though, it's always important to have a clear and specific idea of what it should do. Otherwise, it's easy for us to get confused by trying to solve two problems at once: *what* the function should do, and *how* it should do it. So let's take the first problem first.

It helps to start by writing down a brief sentence or two describing the function's behaviour, ideally with specific examples: given some value as input, what should the function return as a result?

This step forces us to clarify our ideas, makes our assumptions explicit, and gives us a way of checking that the code we'll eventually write is correct.

So what sentence could we write to describe the *specific* behaviour of `Encipher`? We can choose a short message to encipher, and work out the expected result in advance. Here's one possibility, for example:

- `Encipher` transforms “HAL” to “IBM”.

Notice how short and simple this statement is. In particular, it says *what* happens, but it doesn't say anything about *how* it should happen. It doesn't specify a key, or a cipher scheme, or anything like that. It describes the function's behaviour purely in terms of inputs and outputs, treating `Encipher` as though it were a *black box* whose inner workings are completely hidden from the caller.

That's good, because callers shouldn't need to know about a function's implementation, only its behaviour. In fact, as programmers we reserve the right to *change* the implementation later, and so long as it doesn't change the behaviour, everything will continue to work just as well as before.

From behaviour to test

To enforce this implementation-hiding, we'll turn our behaviour sentence into an automated test, so that whatever we do to `Encipher`, we'll always have a reliable (and quick) method of checking that we haven't altered its behaviour in any user-visible way.

I've given you a head start by writing a Go test for `Encipher` that expresses the behaviour sentence we just wrote. You'll find it in the [example repository](#) for this book, and here it is:

```
package shift_test

import (
    "bytes"
    "testing"

    "github.com/bitfield/shift"
)

func TestEncipherTransformsHALToIBM(t *testing.T) {
    t.Parallel()
    input := []byte("HAL")
    want := []byte("IBM")
    got := shift.Encipher(input)
    if !bytes.Equal(want, got) {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

([Listing shift/1](#))

Choose your own adventure

Throughout this book, we'll be working our way gradually towards a full implementation of the shift cipher, and associated programs. If you like, you can join in right away and tackle the various code challenges as we encounter them. If so, read on for instructions on setting up your own Go project to hold your solutions.

Alternatively, you may prefer to read through the whole book first, focusing on understanding, then come back and work through the challenges in order. If you'd like to do it this way, skip to the next section ("A first test"), and return to this point later, when you're ready to start coding.

Here's how to set up a new project for the shift cipher. Create a new folder on your computer to work in (for example, `shift`). Open a terminal and, within that folder, run:

```
git init
go mod init github.com/YOUR_GITHUB_ID/shift
```

Getting the module name (the argument to `go mod init`) right is important, because otherwise people won't be able to import your package. Replace `YOUR_GITHUB_ID` with your GitHub username.

Create a new file named `shift_test.go` and paste in the test code from listing [shift/1](#). Change the import path from:

```
"github.com/bitfield/shift"
```

to:

```
"github.com/YOUR_GITHUB_ID/shift"
```

Now you're ready to start work. Let's start by reading through this test to see what it does.

A first test

If you've read [For the Love of Go](#), or if you already have some experience with Go programming, I'm sure you'll be able to follow the test code just fine. But let's walk through it anyway, just as a refresher.

Anatomy of a test

First, all Go files begin with a package declaration saying which package the file is part of. In this case, it's the `shift_test` package.

Next, we import some standard library packages we'll need: `bytes` and `testing`.

Because we want to test functions in the `shift` package, we need to import that too, under its full import path: `github.com/YOUR_GITHUB_ID/shift`.

Next, all test functions in Go have names beginning with the word `Test`, and this one is no exception. We could name it anything we like (as long as it begins with `Test`). Here we've chosen to use the behaviour sentence itself as the name (using "[camel case](#)" to avoid spaces, which aren't allowed in Go function names).

We call `t.Parallel` to tell Go that it's okay to run this test in parallel with others (there aren't any others yet, but that will soon change).

Then we set up our `input` to be the slice of bytes representing the string "HAL", and also we establish in advance the result we want when this is enciphered: namely, the slice of bytes "IBM".

In the middle of our test sandwich is the meat: we call the `shift.Encipher` function with our `input`, and save the result as what we got.

When should the test fail?

All this is important, but if we stopped here, it wouldn't really be a useful test. To be useful, a test has to be able to *fail*, and the only question for us is when that should happen.

Well, you know the answer to that: the test should fail when `got` is not equal to `want`. The `bytes.Equal` function will tell us this. If `want` and `got` *are* equal, then everything's fine: we just reach the end of the test function and exit.

On the other hand, if `want` and `got` are different, then the value of `got` must be something we didn't expect. That would mean we have a bug in `Encipher` (or possibly in the test... or even both). Whatever the cause, we'd like to know what we actually *did* get, so we print `got` using `t.Errorf`, along with a reminder of what it should have been (`want`).

That all sounds a bit exhausting, but it's a very simple idea: call the function with some input, compare the result with what you expected, and fail the test if they're not the same. Almost all tests work this way, and the only important things that vary are the inputs and expectations.

The world's greatest bug detector

Great! We have a test. So what's next? Should we go ahead and write `Encipher` now and check if the test passes?

Well, we could, but there's something else we should do first. We should *see the test fail*. What's the point of that?

Well, what is a test, actually? Yes, it's a kind of executable specification of what the function should do, and a way of verifying that the implementation conforms to that spec. But there's another way to think about tests: as *bug detectors*.

Imagine you buy a smoke detector for your home, and the sales droid assures you that it's the finest smoke detector available on the market today. It looks beautiful, it has a long battery life, wi-fi integration, and it literally has bells and whistles.

All of that is wonderful, but it doesn't tell you the one important thing you need to know: *will it sound the alarm when there's a fire?* Because if it doesn't, none of the other features matter at all. And for all you know, it's just a fancy box with no works inside. So how can the sales droid prove to you that it really does detect fires?

Only by actually lighting a fire underneath it and seeing if the alarm goes off, right? And exactly the same thing applies to our test. As long as it's passing, that's very nice, but it tells us nothing about whether it would actually detect bugs in `Encipher`. It could just be the equivalent of an empty box.

To know whether it would really detect a fire, we need to light one. Or rather, if it's supposed to detect bugs, let's put a bug in `Encipher` *on purpose* and see whether the test catches it.

A function about nothing

What bug should we try? Well, probably the simplest thing that could be wrong with `Encipher` is that it does nothing at all! We'd definitely want to detect that, and this test *looks* like it should.

So let's write just such a *null implementation* of `Encipher` and see what happens. Something like this would be fine:

```
func Encipher(plaintext []byte) (ciphertext []byte) {
    return nil
}
```

([Listing shift/1](#))

That's clearly wrong, I hope you'll agree. Whatever the input, `Encipher` always just returns `nil`. If we can't detect *this* bug, then we don't have much of a bug detector, do we?

Here goes:

```
go test
```

```
--- FAIL: TestEncipherTransformsHALToIBM (0.00s)
    shift_test.go:16: want "IBM", got ""
```

Good news, everyone! The bug detector is working—or rather, we've shown that it detects at least *this* bug. There may be others it won't detect. But this is a very good start.

Cutting code

Now it's over to you for the next step: write a working version of `Encipher` that passes the test.

Throughout this book, you'll see a number of code goals for you to achieve, marked with the word **GOAL**, like this:

GOAL: Get the test passing.

When you see this, stop reading at that point and see if you can figure out how to solve the problem. You can try to write the code and check it against the tests, or just think about what you would do.

If you reckon you have the answer, or alternatively if you've got a bit stuck and aren't sure what to do, you can then read on for a **HINT**, or read on even further for the complete **SOLUTION**.

So, go ahead and see if you can get this test passing!

HINT: Don't worry if you find this challenge hard, or even impossible at first. Everyone feels this way in the beginning. When you're not fluent in the language, expressing even the simplest ideas is a struggle, but it will get easier. Just keep practising.

We know, in principle, what `Encipher` should do: it should add one to the value of every byte in its input, and return the resulting slice. Could we write that down as a sequence of steps in *pseudo-code*: something that looks a bit like Go, but is actually just structured English? Let's try.

1. Create a `[]byte` variable to hold the result (let's call it `ciphertext`, since that's what it will be).
 2. Take each byte of the input slice (let's call it `plaintext`) in turn.
 3. Add 1 to this value and assign the result to the corresponding byte of `ciphertext`.
 4. When done, return `ciphertext`.
-

SOLUTION: We can translate the first step of our pseudo-code directly into Go, using the built-in `make` function to create a new slice of a given length, and the built-in `len` function to tell us how long `plaintext` is:

```
ciphertext := make([]byte, len(plaintext))
```

The second step is a bit more complicated; we need a *loop*, and in Go that means a `for` statement. When we want to loop once for every element of a slice, we can use the `range` operator, like this:

```
for i, b := range plaintext {
```

The opening curly brace `{` opens a new code block, and anything inside it will be executed once for every byte in `plaintext`. Each time, the `i` variable will hold the *index* value identifying the byte's position within the slice: 0, 1, 2, 3, and so on. The `b` variable will hold the byte value itself.

The third step is to add one to the current byte value, and set the corresponding byte in `ciphertext` to the result. That's pretty straightforward:

```
ciphertext[i] = b + 1
```

And once this loop is done, then we have the `ciphertext` result we want, so we can return it. The complete function looks like this:

```
func Encipher(plaintext []byte) (ciphertext []byte) {
    ciphertext = make([]byte, len(plaintext))
    for i, b := range plaintext {
        ciphertext[i] = b + 1
    }
    return ciphertext
}
```

([Listing shift/2](#))

The smarty-pants answer

How does your solution compare with this version? It might look quite different, and that's okay. I'm not saying that this is the only way to write `Encipher`, or even that it's the best way. It's *a* way, and it passes the test, so it is by definition correct (if we're using the test as the definition of correctness, which I'm suggesting we should).

If your version of `Encipher` passes the test, then it's also correct, even if it works a totally different way. If, on reflection, you prefer my version, because it's shorter, clearer, or more efficient, feel free to *refactor* your code to look more like this (make sure it still passes the test, though).

On the other hand, if *your* version seems better, then it probably is. Well done! In particular, there's a very short and elegant way to write this function that also passes the test:

```
func Encipher(plaintext []byte) []byte {
    return []byte("IBM")
}
```

At first glance, you might have a variety of reactions to this solution: surprise, amusement, dissatisfaction, even annoyance. Perhaps you feel that *something* is wrong with this, but it's not easy to put into words. Yet, as hard as you search for bugs in this implementation, you won't find any. It passes the test, and that's that.

We're gonna need a bigger test

You're right though: *something* is wrong, but not with the `Encipher` function. The problem is with the test!

Or rather, there's nothing actually wrong with the test either. It verifies that, when given "HAL" as input, `Encipher` returns "IBM", and indeed that's the case. The test faithfully asserts the behaviour we wrote down to describe what we wanted `Encipher` to do.

On reflection, though, it would seem that we haven't quite finished specifying that behaviour. What we *meant*, of course, was that `Encipher` should transform any given input according to the cipher scheme we have in mind. But we didn't *say* that, exactly. We just said that it should turn "HAL" into "IBM", and nothing about what it should do given *other* inputs.

Yes, the short version of `Encipher` turns "HAL" into "IBM", but it would also produce the same ciphertext for *any* input. For example, "HEY BOB U UP", or "SURE BAE COME ON OVER" would also be enciphered as "IBM". And clearly that's not a useful cipher scheme.

A table of cases

If we're not allowed to say anything about the *workings* of `Encipher`, though, how can we express the requirement that different inputs should produce different ciphertexts?

One thing we can do is to *combine* our existing requirement ("HAL" becomes "IBM") with a slightly different requirement. Something like this, perhaps:

- `Encipher` transforms "ADD" to "BEE".

Combining cases

Sure, *by itself* this is no more use than what we already had. We can imagine a version of `Encipher` that simply always returns "BEE", for example, which isn't a big improvement on one that always returns "IBM", even though it would pass this test case.

But the key point is that by *combining* these two cases, we eliminate that option. If we test *both* "ADD" and "HAL", no brain-dead version of `Encipher` can pass by returning a fixed result.

Yes, we can imagine a *perverse* version that would, like this:

```
// Prove me wrong, tests. Prove me wrong.  
func Encipher(plaintext []byte) []byte {  
    switch string(plaintext) {  
        case "HAL":  
            return []byte("IBM")  
        case "ADD":  
            return []byte("BEE")  
    }  
    return nil  
}
```

But that's actually longer than the correct version, and if you're going to go to *this* much trouble, you might as well just write a version of `Encipher` that really works.

A suspiciously similar test

Let's add this new test, then:

```
func TestEncipherTransformsADDTоБEE(t *testing.T) {
    t.Parallel()
    input := []byte("ADD")
    want := []byte("BEE")
    got := shift.Encipher(input)
    if !bytes.Equal(want, got) {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

To pass this *and* the `HALToIBM` test, it's clear we'll need the more sophisticated version of `Encipher`: the one that actually does something. And, indeed, that satisfies both of the tests.

But do we really need two separate, but very similar test *functions*, just to check two different input *cases*? Since everything about these two tests is identical except for `input` and `want`, it seems like it would be shorter, clearer, and simpler to check both cases in a single test.

So let's do just that.

A test case struct

This kind of arrangement is called a *table test*, and it follows a fairly straightforward pattern. First, we set up some kind of struct type to hold each of our test cases:

```
type testCase struct {
    input, want []byte
}
```

Then we can give a list of cases to consider:

```
tcs := []testCase{
{
    input: []byte("HAL"),
    want:  []byte("IBM"),
},
{
    input: []byte("ADD"),
    want:  []byte("BEE"),
},
}
```

Looping over the cases

As usual in Go when dealing with a slice of things, we'll use a `for ... range` loop to check each case in turn. The body of that loop will do what our previous tests did: call `Encipher` with the test input and check the result.

Here's the complete test so far, then:

```
func TestEncipherTransforms(t *testing.T) {
    t.Parallel()
    type testCase struct {
        input, want []byte
```

```

    }
    tcs := []testCase{
        {
            input: []byte("HAL"),
            want:  []byte("IBM"),
        },
        {
            input: []byte("ADD"),
            want:  []byte("BEE"),
        },
    }
    for _, tc := range tcs {
        got := shift.Encipher(tc.input)
        if !bytes.Equal(tc.want, got) {
            t.Errorf("want %q, got %q", tc.want, got)
        }
    }
}

```

Both cases should still pass, since we haven't changed `Encipher` at all, so let's check. Yes, they pass, so that tells us our test refactoring was successful.

Lighting a fire under the test

But we're not quite done yet. However beautifully factored, the test is no use unless it fails in the right way when a bug is present. So let's put a bug in `Encipher` and see what the test does:

```

func Encipher(plaintext []byte) []byte {
    return nil
}

```

Here's the result:

```
--- FAIL: TestEncipherTransforms (0.00s)
shift_test.go:27: want "IBM", got ""
shift_test.go:27: want "BEE", got ""
```

Hmm. Yes, it's failing, but the output messages aren't as helpful as they could be. For example, the first failing case says that it wants "IBM", but for what *input*? Similarly, we don't know what plaintext is supposed to produce "BEE" (at least, we can't tell that without reading the test itself).

This didn't matter before, because the input was included in the *name* of the test:

```
--- FAIL: TestEncipherTransformsHALToIBM (0.00s)
shift_test.go:16: want "IBM", got ""
```

Now that this test checks multiple cases, its name has been reduced to just `TestEncipherTransforms`, and that's less helpful. What can we do?

Introducing subtests

We could transfer the information about the `input` to the failure message itself, but there's a neater way to do this: we can use a *subtest*.

Subtests and `t.Run`

Subtests are a way of grouping together similar cases within a single Go test function, and that sounds like exactly what we want. So how do we create a subtest?

That's easy: we call the `t.Run` method:

```
name := fmt.Sprintf("%s to %s", tc.input, tc.want)
t.Run(name, func(t *testing.T) {
    got := shift.Encipher(tc.input)
```

```
    if !bytes.Equal(tc.want, got) {
        t.Errorf("want %q, got %q", tc.want, got)
    }
})
```

`t.Run` takes the *name* of the subtest as a string; in this case, we've constructed a name like "HAL to IBM".

The second thing it takes is a function with this signature:

```
func (*testing.T)
```

What's that? It's the subtest function itself. Just like a regular Go test, it takes a `*testing.T` that can be used to fail the test.

Here's what the test looks like with this change, then:

```
func TestEncipherTransforms(t *testing.T) {
    t.Parallel()
    type testCase struct {
        input, want []byte
    }
    tcs := []testCase{
        {
            input: []byte("HAL"),
            want:  []byte("IBM"),
        },
        {
            input: []byte("ADD"),
            want:  []byte("BEE"),
        },
    },
}
```

```

    }

    for _, tc := range tcs {
        name := fmt.Sprintf("%s to %s", tc.input, tc.want)
        t.Run(name, func(t *testing.T) {
            got := shift.Encipher(tc.input)
            if !bytes.Equal(tc.want, got) {
                t.Errorf("want %q, got %q", tc.want, got)
            }
        })
    }
}

```

A litany of failure

In our case, the subtest function body is just the code we already had, to call `Encipher` and compare `want` with `got`. There's no difference in the logic, but now when the test fails, we get more information thanks to the subtests:

```

--- FAIL: TestEncipherTransforms (0.00s)
    --- FAIL: TestEncipherTransforms/HAL_to_IBM (0.00s)
        shift_test.go:28: want "IBM", got ""
    --- FAIL: TestEncipherTransforms/ADD_to_BEE (0.00s)
        shift_test.go:28: want "BEE", got ""

```

Note that each case now shows as a separate failure, and the name of the parent test and its subtest are combined using a slash:

```
FAIL: TestEncipherTransforms/HAL_to_IBM
```

Adding more cases

Great. If we were to amuse ourselves by turning this back into an English sentence describing the required behaviour, it would read:

- `Encipher` transforms “HAL” to “IBM”.

Which is what we started with, but we now have a concise way of adding a bunch of other, similar cases. Let's do that:

```
tcs := []testCase{
{
    input: []byte("HAL"),
    want:  []byte("IBM"),
},
{
    input: []byte("ADD"),
    want:  []byte("BEE"),
},
{
    input: []byte("ANA"),
    want:  []byte("BOB"),
},
{
    input: []byte("INKS"),
    want:  []byte("JOLT"),
},
{
    input: []byte("ADMIX"),
    want:  []byte("BENJY"),
},
```

It's fun to think of English words that should produce other English words when enciphered in this way (see if you can find more, or write a program to find them for you).

But, since we're saying `Encipher` should operate just as well on arbitrary numeric data, let's include some:

```
{  
    input: []byte{0, 1, 2, 3, 255},  
    want:  []byte{1, 2, 3, 4, 0},  
},
```

Note the sneaky inclusion of 255 as an input value. That's not random; we'd like to make sure that this edge case wraps around back to zero when enciphered.

A working table test for `Encipher`

Here's the complete, updated test:

```
func TestEncipherTransforms(t *testing.T) {  
    t.Parallel()  
    tcs := []struct {  
        input, want []byte  
    }{  
        {  
            input: []byte("HAL"),  
            want:  []byte("IBM"),  
        },  
        {  
            input: []byte("ADD"),  
            want:  []byte("BEE"),  
        },  
        {  
            input: []byte("ANA"),  
            want:  []byte("BOB"),  
        },  
        {  
            input: []byte("BOB"),  
            want:  []byte("ANA"),  
        },  
    }  
    for _, tc := range tcs {  
        t.Run(fmt.Sprintf("%s-%s", tc.input, tc.want), func(t *testing.T)  
        {  
            runTest(t, tc.input, tc.want)  
        })  
    }  
}
```

```

        input: []byte("INKS"),
        want:  []byte("JOLT"),
    },
    {
        input: []byte("ADMIX"),
        want:  []byte("BENJY"),
    },
    {
        input: []byte{0, 1, 2, 3, 255},
        want:  []byte{1, 2, 3, 4, 0},
    },
}
for _, tc := range tcs {
    name := fmt.Sprintf("%s to %s", tc.input, tc.want)
    t.Run(name, func(t *testing.T) {
        got := shift.Encipher(tc.input)
        if !bytes.Equal(tc.want, got) {
            t.Errorf("want %q, got %q", tc.want, got)
        }
    })
}
}

```

[\(Listing shift/3\)](#)

Notice that we simplified things slightly by not bothering to create a named type definition for the struct here. Instead, we specify it with an *anonymous type literal*, as part of the assignment to `tcs`:

```

tcs := []struct {
    input, want []byte
}{


```

```
// ... data goes here  
}
```

This is common in table tests, as you can imagine, because we *always* have some test case type, and it's usually obvious from context that that's what it is, so it doesn't need a name.

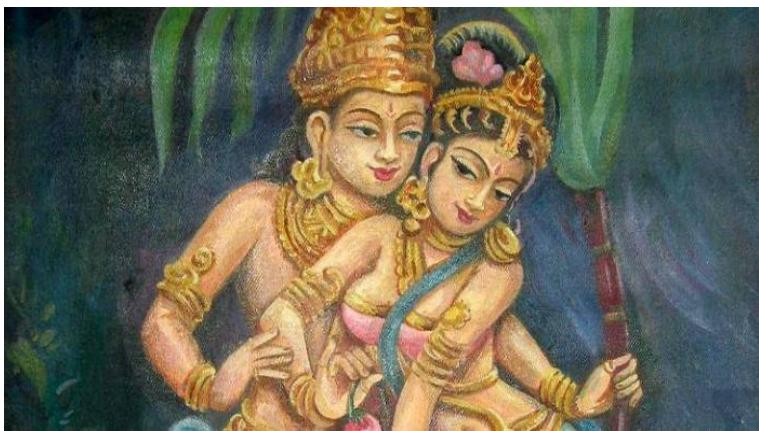
If we restore the `Encipher` function to its working version, we should find that it passes all these test cases, and indeed it does. Reassuring!

2. Enciphering

The Kāma-sūtra recommends that women should study 64 arts, such as cooking, dressing, massage and the preparation of perfumes. The list also includes some less obvious arts, namely conjuring, chess, bookbinding and carpentry.

Number 45 on the list is mlecchita-vikalpā, the art of secret writing, advocated in order to help women conceal the details of their liaisons.

—Simon Singh, ["The Code Book"](#)



We now have a program that applies our simple shift cipher to arbitrary data. Or do we?

Well, not so fast. Actually, although we've written some useful code, we don't yet have a *program*, in the sense of some command we could run that would actually encipher real data. Let's see if we can add that now.

A simple enciphering filter

First, how should it behave? The simplest user interface we can imagine is for the program to act as a *filter*: it just reads from standard input and writes to standard output. Many familiar tools work this way, and it means we can include our enciphering program as part of a *pipeline*, maybe including other transformations.

When Alice wants to send a secure message to Bob, she can write the plaintext in her favourite text editor, and then “pipe” that text into our enciphering tool. Its output will be the ciphertext message she transmits to Bob (and we’ll worry about how he’s going to decipher it later). First let’s take care of the enciphering.

A command package

You probably know that in order to produce a command—that is, an *executable binary*—from a Go program, we need to compile a package named `main`. Let’s create a new subfolder for this package, named after the command it implements:

```
mkdir -p cmd/encipher
```

And now we’ll add a `main.go` file to this folder. This is the “glue code” that connects our `shift` package with the user’s command line, and turns it into a useful tool. So what does it need to do?

Garbage in, garbage out

Not much, as it happens. It just needs to read data from standard input, pass it to `Encipher`, and write the resulting enciphered data to standard output, and that’s straightforward:

```
package main

import (
    "fmt"
    "io"
    "os"
```

```
"github.com/bitfield/shift"  
)  
  
func main() {  
    plaintext, err := io.ReadAll(os.Stdin)  
    if err != nil {  
        fmt.Fprintln(os.Stderr, err)  
        os.Exit(1)  
    }  
    ciphertext := shift.Encipher(plaintext)  
    os.Stdout.Write(ciphertext)  
}
```

([Listing shift/3](#))

Because we need to import the `shift` module, we have to tell Go where to find it. Change the “`github.com`” import path in your `main.go` to whatever you declared your module name to be. For example:

```
"github.com/YOUR_GITHUB_ID/shift"
```

For testing, we can use the `echo` command and the shell’s “pipe” operator (`|`) to pass some text into the program’s standard input:

```
echo "Hello, world" | go run ./cmd/encipher
```

Ifmmp-!xpsme

Nice! Let’s try a longer message:

```
echo "THESE PRETZELS ARE MAKING ME THIRSTY" | go run  
./cmd/encipher
```

UIFTF!QSFU[FMT!BSF!NBLJOH!NF!UIJSTUZ

Looks pretty incomprehensible, unless you know the cipher scheme (and the key). But, on closer inspection, there are some interesting patterns in the ciphertext.

All about Eve

We've already met Alice and Bob, the traditional sender and receiver in the cryptographic literature. They want to have a private conversation, but private from whom?

We'll assume that Alice and Bob can't meet in person, or otherwise arrange some kind of communication channel whose security is guaranteed. If they could, they wouldn't need cryptography. So Alice must be able to protect her messages in such a way that they can be sent, effectively, in public.

To put it another way, if some eavesdropping third party should get hold of the ciphertext, they shouldn't be able to learn anything about the enciphered message—at least, if the cipher is any good.

This new addition to our cast of characters is traditionally named “Eve” (yes, computer scientists are just as prone to dad jokes as physicists). So let's put ourselves in Eve's place for a moment and see how she might go about analysing the intercepted ciphertext.

The aim of this *cryptanalysis* is to figure out the key, or otherwise to recover the plaintext. If that's not possible, Eve might still be able to exploit weaknesses in the key or cipher, or in Alice and Bob's way of using them, to learn at least *something* useful about the conversation.

Reverse engineering

One way that Eve could analyse the ciphertext is to look at the relative frequency of the different letters in it. Here's our message again:

UIFTF!QSFU[FMT!BSF!NBLJOH!NF!UIJSTUZ

Can you see any interesting patterns? Let's start by looking for the most common letter in the ciphertext, and assume that it represents E, which is the most common letter in English text.

There are six occurrences of F, more than any other letter, so perhaps the cipher transforms E to F. That would be a key—a shift value—of 1, so let's apply the inverse transformation and see what we get:

THESE PRETZELS ARE MAKING ME THIRSTY

Oh dear. Our seemingly-impenetrable cipher wasn't actually all that hard to crack, given a couple of sensible guesses, and a single well-known fact about the English language. If Alice and Bob want to keep their snack-chat from Eve's ears, they'll have to work a little harder.

Variable keys

And, if Eve happens to get hold of a copy of Alice's enciphering program, she doesn't even need to break out the frequency analyser. With the current version of the program, there can *be* only one key: namely, 1.

This is officially the world's weakest cipher! So let's fix that problem first, by adding a way for Alice to select a different key.

Adding a keyhole

GOAL: Modify the test so that it expects `Encipher` to take an additional argument specifying the cipher key. Make sure the test cases include different key values other than 1.

Don't worry about *implementing* this change yet, just update the test as though `Encipher` already worked this way. You won't be able to run it just

yet, because the code won't compile—this is expected, and we'll deal with that after we've sorted out the test.

HINT: We're saying that the key will be different for each test case, so that sounds like we'll need a new field on our test case struct type.

We'll also need some new cases, testing a range of different keys. Finally, we'll need to modify the test logic itself to *pass* each key to `Encipher`. See what you can do!

SOLUTION: Let's take things step by step. First, since we'll be using different keys for different test cases, we'll need to add a new key field to our test case struct:

```
tcs := []struct {
    key        byte
    input, want []byte
}
```

We could have used `int` for the key, of course, but since we'll be adding this value to a `byte`, it makes sense to use a `byte` here, too. And, since there are only 255 useful keys, we don't need anything bigger than a `byte` to store them in.

Why are there only 255 useful keys, by the way? Well, the maximum value you can store in a `byte` is 255. If you try to increment a `byte` value beyond that, it just flips around back to zero, as one of our test cases demonstrates.

If you include zero, that gives us 256 possible values for `key`. But enciphering with a key of zero would leave the plaintext unchanged: in other words, zero is a *weak key*. So our cipher scheme is currently limited to 255 useful keys.

New keys, new cases

Now let's give some test cases, using a few different keys:

```
tcs := []struct {
    key        byte
    input, want []byte
}{

{
    key:    1,
    input: []byte("HAL"),
    want:   []byte("IBM"),
},
{
    key:    2,
    input: []byte("SPEC"),
    want:   []byte("URGE"),
},
{
    key:    3,
    input: []byte("PERK"),
    want:   []byte("SHUN"),
},
{
    key:    4,
    input: []byte("GEL"),
    want:   []byte("KIP"),
},
{
    key:    7,
    input: []byte("CHEER"),
    want:   []byte("JOLLY"),
}
```

```
},
{
    key:    10,
    input: []byte("BEEF"),
    want:   []byte("LOOP"),
},
}
```

Since the key is now part of the test case, we should report it as part of the subtest name:

```
name := fmt.Sprintf("%s + %d = %s", tc.input, tc.key, tc.want)
```

Finally, we need to pass the key as the second argument to `Encipher`:

```
got := shift.Encipher(tc.input, tc.key)
```

So here's what we end up with:

```
func TestEncipherTransforms(t *testing.T) {
    t.Parallel()
    tcs := []struct {
        key         byte
        input, want []byte
    }{
        {
            key:    1,
            input: []byte("HAL"),
            want:   []byte("IBM"),
        },
    }
}
```

```
        },
        {
            key:    2,
            input: []byte("SPEC"),
            want:  []byte("URGE"),
        },
        {
            key:    3,
            input: []byte("PERK"),
            want:  []byte("SHUN"),
        },
        {
            key:    4,
            input: []byte("GEL"),
            want:  []byte("KIP"),
        },
        {
            key:    7,
            input: []byte("CHEER"),
            want:  []byte("JOLLY"),
        },
        {
            key:    10,
            input: []byte("BEEF"),
            want:  []byte("LOOP"),
        },
    }
}

for _, tc := range tcs {
    name := fmt.Sprintf("%s + %d = %s", tc.input, tc.key,
    tc.want)
    t.Run(name, func(t *testing.T) {
        got := shift.Encipher(tc.input, tc.key)
        if !bytes.Equal(tc.want, got) {
```

```
        t.Errorf("want %q, got %q", tc.want, got)
    }
})
}
}
```

([Listing shift/4](#))

We know this won't work yet, since we haven't modified `Encipher` to take the key parameter, but let's just see where we are:

```
go test
```

```
./shift_test.go:51:36: too many arguments in call to shift.Encipher
have ([]byte, byte)
want ([]byte)
```

Makes sense: we're now passing the key to `Encipher`, but it's not ready to *take* the key. Let's fix this error in the next step.

Compiler complaints

That wasn't too difficult in principle, was it? The only obstacle is psychological: this test is now a work of imagination, just as our first one was. Imagination is required, not because `Encipher` doesn't exist, but because it doesn't yet take a second argument.

Let's address that now.

GOAL: Modify `Encipher` so that the test compiles, but fails.

Don't fix `Encipher` so that it passes the updated test: we're not there yet. We need to see how our bug detector performs when we know `Encipher` doesn't work. In order to do that, though, the test needs to compile. So your goal here is only to fix the compile error, not to get the test passing.

When you’re done, you’ll see lots of test failures, and we’ll deal with those shortly. For now, focus on eliminating the compile error.

HINT: The Go compiler gives very precise and detailed error messages when something doesn’t compile. These can be hard to read at first, but it’s worth persevering.

The first and most important information in the error message is the exact file, line number, and character position where the error occurred. Use this information to find the offending line in your editor.

Then read the rest of the message carefully. Here it is again:

```
too many arguments in call to shift.Encipher
    have ([]byte, byte)
    want ([]byte)
```

Pay particular attention to the “have X, want Y” part. Go is telling you what it expected to see under the circumstances (“want”), but what you actually *wrote* (“have”) doesn’t match.

Stochastic debugging

It’s tempting, when you see a complicated-sounding error message like this, to just let your eyes slide away from it, and start frantically changing bits of your code at random, to see if the error goes away. We might call this technique “stochastic debugging”, from the Greek for “randomly changing stuff”.

The trouble with stochastic debugging is not that it’s ineffective, but that sometimes it works, thus reinforcing the behaviour. More commonly, though, your changes introduce new and unrelated problems, which in turn obscure the original bug you were trying to fix.

You’ll find it’s more effective in the long run to read the message properly and figure out what’s actually wrong, before you even touch the

problematic code. Learning to *really* read and understand compiler error messages is a superpower which will amaze your fellow developers.

Go always knows what's wrong, and it will even sometimes tell you exactly how to fix it. See if you can apply that idea in this case.

SOLUTION: We're getting a compile error, and that always includes a source file name and line number, so first of all let's locate the problematic line of code. It's this line in the test:

```
got := shift.Encipher(tc.input, tc.key)
```

Here's the error message again:

```
too many arguments in call to shift.Encipher
  have ([]byte, byte)
  want ([]byte)
```

It's complaining that, according to its function signature, `Encipher` only wants one argument (the plaintext), but what we *have* in this function call in the test is two arguments: the plaintext and the key.

That requires us to update the signature of `Encipher`:

```
func Encipher(plaintext []byte, key byte) (ciphertext []byte) {
```

Getting back to green

And that's actually all we needed to do. The test now fails for most of our cases:

```
--- FAIL: TestEncipher (0.00s)
--- FAIL: TestEncipher/SPEC_+_2_=URGE (0.00s)
  shift_test.go:53: want "URGE", got "TQFD"
```

```
--- FAIL: TestEncipher/PERK_+_3=_SHUN (0.00s)
shift_test.go:53: want "SHUN", got "QFSL"
--- FAIL: TestEncipher/GEL_+_4=_KIP (0.00s)
shift_test.go:53: want "KIP", got "HFM"
--- FAIL: TestEncipher/CHEER_+_7=_JOLLY (0.00s)
shift_test.go:53: want "JOLLY", got "DIFFS"
--- FAIL: TestEncipher/BEEF_+_10=_LOOP (0.00s)
shift_test.go:53: want "LOOP", got "CFFG"
```

I quite like this one:

want "JOLLY", got "DIFFS"

Interestingly, the one *passing* case is the original “HAL + 1 = IBM”. On reflection, that makes sense, since `Encipher` currently ignores the selected key and always uses 1 instead. Therefore, we should expect the “key 1” test case to pass, but only by accident.

To make the other cases pass, we’ll need to make a further adjustment to `Encipher` to correct this.

GOAL: Get all cases passing.

HINT: This is a smaller change to `Encipher` than perhaps it seems. Remember, previously we looped over the plaintext, taking each byte and adding 1 to its value to produce the corresponding byte of the ciphertext.

That corresponds to a fixed key of 1, but now instead of using a fixed value, the key is passed *in* as a parameter to `Encipher`. See if you can work out where to use it.

SOLUTION: That was easy, right? I’m sure you figured out that we need to increase each plaintext byte, not by 1 as before, but instead by the value of the new parameter `key`:

```
ciphertext[i] = b + key
```

So here's the complete, updated version of `Encipher` that uses variable keys:

```
func Encipher(plaintext []byte, key byte) (ciphertext []byte) {
    ciphertext = make([]byte, len(plaintext))
    for i, b := range plaintext {
        ciphertext[i] = b + key
    }
    return ciphertext
}
```

([Listing shift/4](#))

Not bad.

Getting the key

Great work on implementing multi-byte keys. Now we need to update the `cmd/encipher` program so that it can use the new facility. How should that work, do you think?

We want Alice to be able to specify the enciphering key as part of the command. So one option would be for her to simply give it as a command-line argument, like this:

```
encipher 5
```

But it's not really clear what that 5 represents in this command, if you don't happen to know. Does it mean encipher the literal message "5"? Encipher something five times? Use Cipher Scheme Five? Who knows.

To make it clearer what this value represents, let's use a named *flag* instead. For example, we could use a flag called `key`.

Now Alice would run the program like this:

```
encipher -key 5
```

Defining a key flag

The standard library's `flag` package does just what we need, so let's import it in `main.go`, and add the following flag definition at the start of `main`:

```
key := flag.Int("key", 1, "shift value")
```

Calling `flag.Int` defines a new flag of type `int`, straightforwardly enough, and the first argument is the *name* of the flag.

On the command line, flags are denoted by a leading dash, and in our case, we want the flag to be specified as “`-key`”. So the name we specify to `flag.Int` should be just `key`, no dash.

The second argument to `flag.Int` is the *default* value for the flag; that is, what value it should take if Alice doesn't happen to specify that flag at all. Let's go with 1 in this example. And the third argument is the help text that will explain to users what the flag is for.

Getting the flag value

Apparently `flag.Int` returns something, so what's that? The documentation tells us that its type is `*int`, that is, a pointer to an `int` value. Can you guess what this is for?

That's right: it's where the actual *value* of the `key` flag will end up. In other words, suppose Alice runs a command like:

```
encipher -key 5
```

In this case, after flags have been parsed, our `*key` variable will contain the value 5. That's reasonable, so now that we've finished defining all the flags we want to take, let's tell the `flag` package to actually parse the program's arguments and set each flag variable to its supplied value:

```
flag.Parse()
```

Now we have Alice's key value stored in `*key`, but there's one further wrinkle before we can use it. `Encipher` takes an argument of type `byte`, but we have an `int` (because there's no `flag.Byte`, unfortunately).

So, we need to convert `*key` to a `byte` before passing it on:

```
ciphertext := shift.Encipher(plaintext, byte(*key))
```

Enciphering with arbitrary keys

Here's the complete `main` function, then, updated to use the `key` flag:

```
func main() {
    key := flag.Int("key", 1, "shift value")
    flag.Parse()
    plaintext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
```

```
    }
    ciphertext := shift.Encipher(plaintext, byte(*key))
    os.Stdout.Write(ciphertext)
}
```

([Listing shift/4](#))

Let's try it out:

```
echo "HEY BOB U UP?" | go run ./cmd/encipher -key 5
```

MJ^%GTG%Z%ZUD

Very nice. And since deciphering is the inverse of enciphering, then it follows that enciphering with a negative key is the same as deciphering with a positive one. In that case, we should also be able to turn this ciphertext back into the original plaintext by enciphering it using the same key with a minus sign, shouldn't we?

Let's try:

```
echo "MJ^%GTG%Z%ZUD" | go run ./cmd/encipher -key -5
```

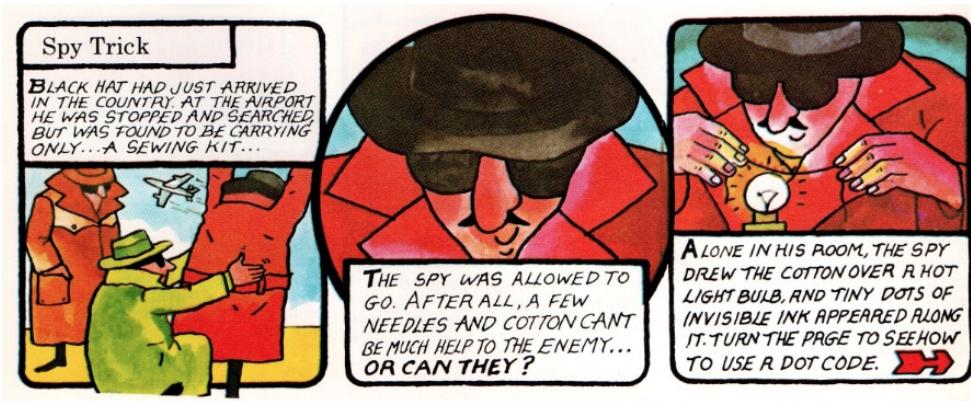
HEY BOB U UP?

Encouraging! So now we can encipher and decipher messages using any key we choose (well, any of the 256 possibilities represented by a single byte value).

3. Deciphering

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files.

—Bruce Schneier, “[Applied Cryptography](#)”



(from [“The KnowHow Book of Spycraft”](#))

So, have we improved the security of our cipher? We saw that the previous version of the program, using a fixed key of 1, was vulnerable to the frequency analysis technique. Eve could start by guessing that the most common ciphertext letter represents E, and go from there.

Let's see if being able to choose a different key avoids this weakness.

Brute force and ignorance

For example, here's the result of enciphering “THESE PRETZELS ARE MAKING ME THIRSTY” with a key of 12:

`TQ_Q, \^Q `fQX_, M^Q, YMWUZS, YQ, `TU^_ `e

It looks pretty much like gibberish. But Eve's frequency analysis attack still works just as well here, it turns out. The most common ciphertext letter is now Q (17th in the alphabet), and if she assumes it represents E (5th in the

alphabet), then the required shift value gives her the key straight away: it's 12.

Dang!

A statistical vulnerability

It's clear that one of the major weaknesses in our cipher is not the choice of key, but the fact that each plaintext letter always produces the *same* letter in the ciphertext. As a result, the characteristic letter frequencies of English (or whatever the plaintext language is) are a dead giveaway.

Of course, there are perfectly legitimate messages where E happens not to be the most common letter. The next most common letters in English are T, A, I, N, O, and S, in that order, and they would also be reasonable guesses for the frequency analyst. Other languages have a slightly different frequency table, but not *that* different.

This is a statistical technique, so the longer the ciphertext Eve has to work with, the more effective it becomes. To make Eve's job harder, therefore, Alice should minimise the amount of data enciphered with a given key. We'll look at one way to do this later on.

However, we have made one important improvement in the strength of our cipher. With a fixed key (whether it's 1 or any other number), the secrecy of our messages would depend entirely on Eve not knowing the cipher scheme we're using. Now that Alice can choose different keys, this is no longer true.

It's still a good idea to keep the details of the cipher secret, but it's no longer such a disaster if they become known to Eve, or anyone else. Alice must, however, keep her key itself a secret, or the cipher is no cipher at all.

In fact, it's always safest to assume that, in the words of Claude Shannon, "the enemy knows the system", and to make sure that such knowledge is as little help to them as possible.

How many possible keys are there?

And there's another, related problem. Because we're working with bytes, there's only so far we can shift them. A single byte can represent values between 0 and 255, so, as we saw earlier, that limits our cipher to just 255 possible useful keys.

A cipher with a small number of keys is weak: if Eve can just *guess* the key within a reasonable number of attempts, then Alice and Bob's chats won't be secure for long.

This becomes more likely if Alice isn't very good at choosing random keys, and instead uses her dog's name, her mom's birthday, or her favourite food as a key. Now if Eve knows something about Alice, or can find out, she has an improved chance of guessing the key. (We'll talk about how Alice could choose better keys later on in this book.)

But if Eve doesn't know who enciphered the message, or even whether or not they have a dog, she's still in with a chance. Even completely uninformed guesses will eventually hit on the right answer, given enough time.

The simplest key-guessing approach, in fact, is known as *brute force*: Eve just tries every possible key. It's not sophisticated, but it is guaranteed to succeed... eventually.

How effective would this be in practice, though? For what values of "eventually" would the cipher really be vulnerable? Let's do a little quick estimation.

How safe is your safe?

Suppose Alice keeps her valuables (her cipher keys, perhaps) in one of those old-timey safes protected by a combination lock, with three single-digit number wheels. You can check my math on this one, but since the combination could be any number between 000 and 999, that gives a total of one thousand possible combinations.

And suppose Cat, a burglar, wants to get into the safe without using explosives (too noisy) or drilling the lock (too slow). All she can do is try to

guess Alice's combination. How much time would Cat need to have a decent chance of accessing the safe?

We should first ask how long it takes Cat to try a particular combination. She has to dial the number in and yank the handle to see if the safe opens. That won't take long, especially if she's trying the combinations in sequence. Let's say it's one second per combination.

So Cat could try all possible combinations—all possible keys, that is—in a little over sixteen minutes. That's not a tremendously long time, especially if the safe is full of your hard-earned cash and valuables.

It gets worse, though, because she probably won't even need to try *every* possibility. On average, she'll find the right combination about half-way through the list of possibilities.

To put it another way, a three-digit combination gives Cat a 50% probability of brute-forcing the safe within eight minutes. (That's one of the reasons why you don't see many safes with combination locks these days.)

Doing it for the crack

What about our shift cipher, then? With a key length of one byte, how many different keys would Eve have to choose from, and how long would it take her to try half of the possibilities?

Just for fun, let's see how long it actually takes to brute-force this key using a Go program. That will give us a baseline to work from, so that we can measure how future improvements to our cipher affect the brute-forcing time.

What would such a program look like? It seems like it wouldn't be too hard to generate each of the possible keys, but how will we know when we've found the *right* key? That's not so straightforward.

Cribs

One option is just to print out the plaintexts resulting from all 255 possible keys, and leave it to Eve to decide if any of them looks like an intelligible message. That's not a bad approach in principle, and we could even imagine trying to detect valid plaintexts automatically in some way. We could look for English words, for example, and consider the plaintext containing the most dictionary words as the one most likely to be the true message.

But let's keep things simple for now by assuming that we have some reliable way to identify the correct key once we find it. One such way is to compare the candidate plaintext against a *crib*: that is, some known fragment of the true plaintext.

For example, if I give you the following ciphertext:

IQSJGT

and I tell you that the plaintext message begins with the letters “GO”, then it’s very easy for you to check your guesses as you’re conducting the brute-force key search. Simply try each key on the first two letters and see if they produce “GO”. If they do, then you’ve cracked it!

Such cribs can be easier to find than you might think. Any stereotypical or predictable text can be a crib. For example, if you were confronted with an enciphered Go source file, you could make a very good guess that the first word of the plaintext would be package. (It might be a build tag, but package is much more likely.)

Indeed, most digital file formats have some kind of predictable header data that could act as a crib, as we’ll see later on. So, assuming we can supply a suitable crib, how can we use it to crack the cipher?

Designing a cracking function

Let’s imagine, then, that we have a function called `crack` that can take a ciphertext and a crib, check various keys against the crib, and return the result. How would it work, in principle? Let’s sketch out some ideas.

We'll need a loop so that we can try each possible key value from 0 to 255. Within that loop, what do we need to do? What does it mean to "try" a key?

Well, we need to use the key to decipher the message, but not all of it; just enough to produce the crib (if we have the right key). So if the message is 100 bytes long, and we have a 10-byte crib, then we only need to try decrypting the first 10 bytes of the message. We can compare the result of this with the crib, and if it's the same, then we very likely have the key.

First, we need Decipher

This is starting to look manageable, but there's a problem: we don't actually have a `Decipher` function yet. And even though we haven't yet attempted to write `crack`, it's clear that it will be much easier to write if we already had `Decipher`. So let's start there.

GOAL: Write a test for a `Decipher` function, in the same way that you did for `Encipher`. This time, you decide what parameters the function should take and what results it should return.

Given that we already have a test for `Encipher`, and we know that deciphering is the exact inverse of enciphering, this isn't too difficult, is it? Let's start with a single test case:

```
func TestDecipherWithKey1TransformsIBMTоХАL(t *testing.T) {
    t.Parallel()
    input := []byte("IBM")
    want := []byte("HAL")
    got := shift.Decipher(input, 1)
    if !bytes.Equal(want, got) {
        t.Errorf("want %q, got %q", want, got)
    }
}
```

Now you can go ahead and write the `Decipher` function yourself:

GOAL: Implement `Decipher`, and check your solution passes the test before reading on.

HINT: If you're not sure what to do, you can always refer to the existing code for `Encipher`, and see if that gives you any ideas. `Decipher` must be very similar in structure, mustn't it? But there's one important difference. Can you see what to change?

SOLUTION: Well, one completely reasonable solution to this problem is to write something very similar to `Encipher`, but that subtracts the key from each byte instead of adding it:

```
func Decipher(ciphertext []byte, key byte) (plaintext []byte) {
    plaintext = make([]byte, len(ciphertext))
    for i, b := range ciphertext {
        plaintext[i] = b - key
    }
    return plaintext
}
```

If this was your answer, it's fine, but can you do better? Is there a shorter, neater, more elegant solution?

GOAL: Do better.

HINT: As we saw earlier, deciphering is very similar to enciphering. In fact, with the shift scheme we're using, it's identical, except that we *subtract* the key from the plaintext byte instead of adding it.

To put it another way, we can think of deciphering as being simply enciphering with a negative key. Does that give you any ideas?

SOLUTION: If deciphering is really just enciphering with a negative key, then we can *call* `Encipher` to implement `Decipher`. This saves duplicating a lot of logic:

```
func Decipher(ciphertext []byte, key byte) (plaintext []byte) {
    return Encipher(ciphertext, -key)
}
```

([Listing shift/5](#))

If you came up with this solution first, great! And if you wrote the longer version first, that's also just fine, because it's the *behaviour* that matters in the end, not the code.

Do we even need a test, then?

Since we already wrote a really thorough table test for `Encipher`, checking the effect of several different keys, should we do the same for `Decipher`, too? What do you think?

One response to this might be “No, that's a waste of time, because `Decipher` is *implemented* in terms of `Encipher`, and we already tested that.”

But there's a subtle mistake here. We said earlier that good tests are concerned with the program's *behaviour*, not its implementation. So we need to treat `Decipher`, again, as a black box, whose inner workings we have no access to.

Sure, *today* it happens to work by calling `Encipher`, but tomorrow we might change it to use looping and subtraction, or something else. And in order to

make that change it would be helpful to have a test that doesn't rely on the current implementation details.

New test, same old table

So, while the argument for not testing `Decipher` more thoroughly is superficially appealing, it doesn't hold up. Let's go ahead and extend the `Decipher` test to cover all the cases we already checked for `Encipher`.

GOAL: Extend the test for `Decipher` to cover the same cases as for `Encipher`.

HINT: It seems a shame to repeat all those cases in the new test, doesn't it? Instead, could we re-use the existing table of cases from `TestEncipher` somehow?

I think we could, if we extract it from the function and make it a package-level variable. That is, we'll define our cases slice outside any function, so that it's in scope for all our test functions.

We'll need to use a `var` statement for this, since assignment using the `:=` operator isn't allowed outside a function. See what you can do.

SOLUTION: By using `var` to define our slice, we can place this code in `shift_test.go`, outside the test functions:

```
var cases = []struct {
    key          byte
    plaintext, ciphertext []byte
} {
{
    key:      1,
    plaintext: []byte("HAL"),
```

```

    ciphertext: []byte("IBM"),
},
{
    key:      2,
    plaintext: []byte("SPEC"),
    ciphertext: []byte("URGE"),
},
{
    key:      3,
    plaintext: []byte("PERK"),
    ciphertext: []byte("SHUN"),
},
{
    key:      4,
    plaintext: []byte("GEL"),
    ciphertext: []byte("KIP"),
},
{
    key:      7,
    plaintext: []byte("CHEER"),
    ciphertext: []byte("JOLLY"),
},
{
    key:      10,
    plaintext: []byte("BEEF"),
    ciphertext: []byte("LOOP"),
},
}

```

[\(Listing shift/5\)](#)

We've tweaked the field names slightly, since `input` and `want` don't really apply anymore: it depends on whether you're testing enciphering or deciphering. Instead, we use `plaintext` and `ciphertext` to make it clear.

Look before you loop

Now we can loop over cases in both tests, since all functions defined in a package have access to variables defined at the package level.

Here are the updated tests for both functions:

```
func TestEncipher(t *testing.T) {
    t.Parallel()
    for _, tc := range cases {
        name := fmt.Sprintf("%s + %d = %s", tc.plaintext, tc.key,
            tc.ciphertext)
        t.Run(name, func(t *testing.T) {
            got := shift.Encipher(tc.plaintext, tc.key)
            if !bytes.Equal(tc.ciphertext, got) {
                t.Errorf("want %q, got %q", tc.ciphertext, got)
            }
        })
    }
}

func TestDecipher(t *testing.T) {
    t.Parallel()
    for _, tc := range cases {
        name := fmt.Sprintf("%s - %d = %s", tc.ciphertext,
            tc.key,
            tc.plaintext)
        t.Run(name, func(t *testing.T) {
            got := shift.Decipher(tc.ciphertext, tc.key)
            if !bytes.Equal(tc.plaintext, got) {
                t.Errorf("want %q, got %q", tc.plaintext, got)
            }
        })
    }
}
```

```
    })  
}  
}
```

([Listing shift/5](#))

A deciphering tool

We already have a command-line tool for enciphering, so, before we move on, let's take this opportunity to add a deciphering tool. We have a working `Decipher` function, so how can we turn this into a command that Bob can run to decrypt Alice's messages?

Let's not complicate encipher

One approach might be to add another flag (`-decipher`, for example) to our existing `encipher` program. But this feels a little awkward, because the user would have to run some command like:

```
encipher -decipher ...
```

That's weird, right? Instead, let's just write a separate `decipher` program. Instead of one complicated program that does two things, we'll have two simple programs, each of which only does one thing. That's easier for us to write, and Bob will likely prefer it, too.

GOAL: Implement the `decipher` command.

HINT: Let's start with where the `main.go` file should fit into our project structure. It's a command, so somewhere under the `cmd` folder makes sense. Since we already have `cmd/encipher`, let's create a new `cmd/decipher` subfolder.

What should the `main` function be? Well, as we've noted before, deciphering is very similar to enciphering, so looking at the `main` function for the `encipher` command should give you a useful starting point. Good luck!

SOLUTION: Something like this should work fine:

```
package main

import (
    "flag"
    "fmt"
    "io"
    "os"

    "github.com/bitfield/shift"
)

func main() {
    key := flag.Int("key", 1, "shift value")
    flag.Parse()
    ciphertext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    plaintext := shift.Decipher(ciphertext, byte(*key))
    os.Stdout.Write(plaintext)
}
```

([Listing shift/5](#))

Nothing fancy here: it's almost exactly the same as the `encipher` program, as we expected. The only difference, indeed, is that we call the `Decipher` function instead of `Encipher`.

Let's give it a try:

```
echo "MJ^%GTG%Z%ZUD" | go run ./cmd/decipher -key 5
```

HEY BOB U UP?

Some people might be worried about the duplication of code between `decipher` and `encipher`. Don't be. Duplication is better than *complication*.

4. Cracking

Have you never known the pleasure and triumph of a lucky guess? I pity you. I thought you cleverer; for depend upon it, a lucky guess is never merely luck. There is always some talent in it.

—Jane Austen, [“Emma”](#)



Now we're ready to make a start on the `crack` function, and as usual the first thing we'll do is to write a test. Over to you.

Testing a Crack function

GOAL: Write a test for `crack`.

HINT: As before, we can re-use the existing set of test cases rather than write new ones. And the structure of the test we want is pretty similar to the previous tests for `Encipher` and `Decipher`: loop over our cases, calling our function with each set of inputs, and checking the results against what we expect.

So start by copying one of the existing tests and modifying it to call our (currently imaginary) `crack` function instead of `Encipher` or `Decipher`.

SOLUTION: Here's a test that would do the job:

```
func TestCrack(t *testing.T) {
    t.Parallel()
    for _, tc := range cases {
        name := fmt.Sprintf("%s + %d = %s", tc.plaintext, tc.key,
            tc.ciphertext)
        t.Run(name, func(t *testing.T) {
            got, err := shift.Crack(tc.ciphertext,
                tc.plaintext[:3])
            if err != nil {
                t.Fatal(err)
            }
            if tc.key != got {
                t.Fatalf("want %d, got %d", tc.key, got)
            }
        })
    }
}
```

([Listing shift/5](#))

As you can see, this is very similar to the existing tests; all we've changed is that we call `crack` for each case, and check the result matches `tc.key`.

We've said `crack` needs to take a crib, as well as the ciphertext to work on. The first three bytes of the plaintext should be plenty (actually, one would be enough, in our case, but let's be generous).

Failure is an option

It looks from this code as though `crack` returns an error along with the key, so what's that about? Surely if we try every possible key, we *must* find one

that works, no?

Well, that's true if the ciphertext is indeed enciphered with a single-byte shift. But, in practice, it might not be. And Eve doesn't necessarily have the luxury of knowing the specific encryption scheme that Alice and Bob are using.

What should `crack` do if it's tried every possible key and still hasn't managed to reproduce the crib? Well, it *could* return a value of zero for the key, meaning "no key found". But this doesn't seem very Go-like.

In Go, we have a predefined `error` type for exactly this kind of situation, so let's use it.

If we want a new behaviour, we need a new test:

```
func TestCrackReturnsErrorWhenKeyNotFound(t *testing.T) {
    t.Parallel()
    _, err := shift.Crack([]byte("no good"), []byte("bogus"))
    if err == nil {
        t.Fatal("want error when key not found, got nil")
    }
}
```

([Listing shift/5](#))

A key-guessing function

Now it's over to you again to write the real version of `crack`.

GOAL: Write a `crack` function that passes your tests.

HINT: This is simple in principle, isn't it? We just need to generate each possible key byte in turn, and pass it to `Decipher`. When the deciphered

result matches the crib, we've found the key, so return it. Otherwise, continue to the next key.

That suggests a loop over the 256 possible byte values. Each time round the loop, we'll pass the candidate key to `Decipher` and see if it produces a plaintext that matches our crib.

Can you see what to do?

SOLUTION: Here's a version that passes the test:

```
func Crack(ciphertext, crib []byte) (key byte, err error) {
    for guess := range 256 {
        result := Decipher(ciphertext[:len(crib)], byte(guess))
        if bytes.Equal(result, crib) {
            return byte(guess), nil
        }
    }
    return 0, errors.New("no key found")
}
```

([Listing shift/5](#))

Note that we don't pass the whole ciphertext to `Decipher`; that would be wasteful, since we only need to check our decryption against the bytes available in the crib.

A command-line cracker

Nice job on writing `crack!` This looks very promising, and it passes all our test cases, so we're now ready to implement the cracking behaviour as a command-line tool that Eve can use to analyse Alice and Bob's intercepted messages.

GOAL: Implement a `crack` command.

HINT: While we *could* do this by extending the existing `decipher` command, that seems a shame, for the reasons we've discussed. Instead, we'll make a separate new command under `cmd/crack`.

Now, since the `crack` function returns the key (if available), our command could simply print this value out. But what would Eve do then? Just run the `decipher` command with the provided key, probably.

What do users want?

Actually, it turns out what Eve wants is not really the key itself; that's just a means to an end. She knows Alice is too smart to re-use the same key for multiple messages.

What Eve wants is the plaintext, so our tool would be more helpful if it also deciphered the message. And we already have the code for that part, since the `decipher` tool does exactly this.

So, our `crack` program will be quite similar to `decipher` in principle, except that we'll need a flag for Eve to pass in the crib, instead of the key.

We can read the ciphertext from standard input as before, use the `crack` function to recover the key, checking our results against Eve's supplied crib, and finally call `Decipher` to produce the plaintext.

Is this enough information for you to get cracking? (Sorry.)

SOLUTION: Here's a program that would do the trick:

```
package main
```

```
import (
```

```

"flag"
"fmt"
"io"
"os"

"github.com/bitfield/shift"
)

func main() {
    crib := flag.String("crib", "", "crib text")
    flag.Parse()
    if *crib == "" {
        fmt.Fprintln(os.Stderr, "Please specify a crib text with
-crib")
        os.Exit(1)
    }
    ciphertext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    key, err := shift.Crack(ciphertext, []byte(*crib))
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    plaintext := shift.Decipher(ciphertext, byte(key))
    os.Stdout.Write(plaintext)
}

```

([Listing shift/5](#))

Crib constraints

We've made the `-crib` flag mandatory, since we don't at the moment have any other way of checking the guessed key. We've talked about some possible ways of doing this without a crib, though, and you might like to explore those in order to build a cracking tool that doesn't need one.

For simplicity, our tool also only accepts a crib that *begins* the plaintext. In other words, we can't use it to crack a ciphertext where we know part of the middle, or the end, for example. The first byte of the crib must be the first byte of the plaintext, or the tool won't work.

Again, if you want to have some fun making this tool more flexible and useful, you might like to think about ways to make it work with a crib that merely occurs *somewhere* in the plaintext, not necessarily at the beginning. This would certainly be slower, because we'd have to decipher the whole text to check each key, but that's okay. Slow deciphering is usually better than no deciphering at all.

Cracking a real ciphertext

Let's try out our new crack tool and see if we can use it to recover the plaintext of an enciphered file. First, let's start with a small piece of text, which we can save to a file named `tiger.txt`:

The tiger appears at its own pleasure. When we become very silent at that place, with no expectation of the tiger, that is when he chooses to appear... When we stand at the edge of the river waiting for the tiger, it seems that the silence takes on a quality of its own. The mind comes to a stop. In the Indian tradition that is the moment when the teacher says, "You are that. You are that silence. You are that."

—Francis Lucille, [“The Perfume of Silence”](#)

([Listing shift/5](#))

Waiting for the tiger

We'll run this through the `encipher` command to create the ciphertext:

```
go run ./cmd/encipher -key 253 <tiger.txt >enciphered.bin
```

We use the .bin extension just to remind ourselves that the output of this process isn't a text file like the one we started with: it's now arbitrary binary data.

Let's have a look at it, though, just to make sure it doesn't reveal anything obvious about the plaintext:

```
cat enciphered.bin
```

```
Qebqfdb^mmb^op^qfqpltkmib^prob+Tebktb_b`ljbsbovpfibkq^qqe^qmi^`b)  
tfqeklbumb`q^qflklcqebqfdb)qe^qfp tebkeb`ellpbpbql^mmb^+++Tebktbpq^  
ka^qqebb badblc qeb of sbot^f qfkdcloqebqfdb) fqpbbjpqe^q qebpfibk`bq^hbp  
lk^nr^ifqv lcfqpltk+Qebjfka`ljb pql^pqlm+FkqebFkaf^kqo^afqflkqe^qfpq  
ejjljbjkqtebkqebqb^`ebop^vp).}.Vlr^obqe^q+Vlr^obqe^q pfibk`b+Vlr^obq  
e^q+.}..}.Co^k`fpIr`fiib).}.QebMbocrjblcPfibk`b.).%
```

Well, it doesn't look like anything to me!

Let's see if we can recover the plaintext with the crack tool, then:

```
go run ./cmd/crack -crib 'The tiger' <enciphered.bin
```

And very quickly:

The tiger appears at its own pleasure. When we become very silent at that place, with no expectation of the tiger, that is when he chooses to appear... When we stand at the edge of the river waiting for the tiger, it seems that the silence takes on a quality of its own. The mind comes to a stop. In the Indian tradition that is the moment when the teacher says, "You are that. You are that silence. You are that."

—Francis Lucille, "The Perfume of Silence"

That's encouraging. Given a suitable crib, we can automatically recover the plaintext from a message enciphered with an unknown key. Great progress!

The devil in the details

We've been careful to ensure from the start that our program should be able to handle any kind of data, not just text, so let's try that too. For example, what if we enciphered an image? Would we still be able to recover the original image data using `crack`?

It doesn't matter what kind of image we use, but just for fun, we'll download this [little devil](#), and save it to a file. Here's what it looks like, using any decent image-viewing program:



Now we'll encipher it:

```
go run ./cmd/encipher -key 99 <devil.png >devil.bin
```

The resulting `devil.bin` file doesn't in any way resemble an image, and indeed most image-viewing software will refuse to open it. We'll see what we can do to recover the original with `crack`, but first, what's our crib?

0x89 is the magic number

Well, without access to the original file, we wouldn't know anything about the *image* data it contains. Luckily, though, most well-known file formats begin with some stereotyped byte sequence that identifies the format (known as a *header*, or more romantically a *magic number*).

For example, all PNG files begin with the hex byte 0x89, followed by the letters "PNG". (If you're not familiar with hex notation, don't worry: we'll return to this later.)

That's a perfect crib, but there's one little hiccup: how can we actually supply these bytes to the `-crib` flag on the command line? The value 0x89 doesn't correspond to any printable character, so we can't type it on the keyboard. What to do?

There's a helpful little Unix utility named `printf` that can solve this problem for us. It prints any string we give it, but within that string we can include *escape sequences*: ways of representing non-printing characters.

Here's how we do that:

```
printf '\x89PNG'
```

This prints `PNG`, as you'd expect, but prefixed by a weird character that looks like a question mark inside a diamond-shaped box (depending on your terminal font). This is the Unicode "replacement character" glyph, meaning that the byte in question doesn't correspond to any printable character, which we already know it doesn't.

Printing the unprintable

So we can't just copy and paste this string straight from the terminal, because that special glyph won't be translated back into the byte 0x89.

Instead, we can embed the call to `printf` directly into our `crack` command line:

```
go run ./cmd/crack -crib $(printf '\x89PNG') <devil.bin  
>plain.png
```

The `$(...)` syntax is an instruction to the shell: run the command inside the parentheses first, then substitute the result into the full command line.

Sure enough, the output `plain.png` file is byte-for-byte identical with the original `devil.png`. If we open it in an image viewer we will see the little devil, unharmed by his journey through our shift cipher:

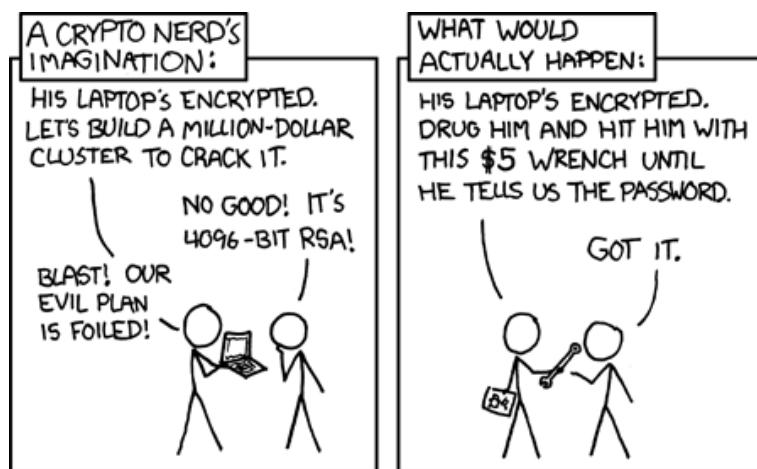


5. Keys

Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations.

(They are also large, expensive to maintain, difficult to manage, and they pollute the environment. It is astonishing that these devices continue to be manufactured and deployed. But they are sufficiently pervasive that we must design our protocols around their limitations.)

—Charlie Kaufman et al, [“Network Security: Private Communication in a Public World”](#)



Okay, let's get back to work: we still have a few more things to add before we're able to deal with realistic ciphers.

Lost in keyspace

We talked earlier about cipher strength in the context of brute-forcing, and it's clear that the smaller the number of possible keys, the easier it will be for Eve to defeat the cipher.

And we already know that our single-byte shift cipher is pretty weak in this respect, because it just doesn't allow for enough possible keys. The proof is

that we can write a (fairly) simple Go program to crack any ciphertext in a few microseconds. So this cipher scheme, while easy to reason about, isn't strong enough to protect any sensitive information.

The space of all possible keys

What would *make* it stronger, then? One obvious way would be to make the key longer than a single byte, thus making the space of all possible keys—the *keyspace*, we say—bigger. The bigger the keyspace, the longer a brute-force search of it takes.

Adding a fourth number wheel to Alice's office safe, for example, would give her ten times as many possible keys. She can set the new wheel to ten different digits for each possible arrangement of the original three wheels, increasing the keyspace from 1,000 to 10,000 combinations.

Now Cat the burglar would need to remain undisturbed in Alice's office twiddling the safe for (on average) eighty minutes, and there's a much greater risk of her being caught than when it only took eight minutes.

How long should the key be?

In our shift cipher, similarly, we could make the key two bytes instead of one. By the same argument as for the safe, that gives us $256 \times 256 = 65,536$ possible keys. It will therefore take 256 times longer to brute-force, which is a big improvement, perhaps surprisingly so.

You might quite reasonably have guessed, for example, that doubling the key length would make Alice's messages twice as secure, but it's very much better than that.

In fact, every extra byte she adds to the key makes the keyspace (and thus the average crack time) 256 times bigger. This is why reasonably long passwords are so astoundingly much more secure than short ones.

For example, a seven-character password can be cracked (by 2020s computers) in about six minutes, but a fourteen-character one would take

over 200 million years! That's the counterintuitive power of exponential growth (and also the reason why you should use long passwords).

So we won't need to make our key *too* many bytes longer to get a really worthwhile increase in the strength of our cipher. 32 bytes, or 256 bits, is the biggest practical key size in current use, and we're unlikely to need larger keys for the foreseeable future.

Even with the help of [Moore's Law](#), it will take Eve a long time to double her available computing power. But when that happens, all Alice needs to do is add one more bit to her keys, and she's put Eve right back where she started. It's an arms race that Eve can't win.

Doubling computer power is also a lot more expensive than just making our keys one bit longer, as you can imagine. So simply increasing the key length is by far the most cost-effective cipher improvement we can make, at least until we reach the point where brute-forcing is no longer the quickest way to break the cipher.

A keyspace the size of the universe

How would lengthening our keys to 32 bytes affect the size of our keyspace, then? Well, each extra byte makes it 256 times bigger, so our keyspace now becomes 256^{32} , or 2^{256} , which is an absolutely gigantic number:

115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,5
64,039,457,
584,007,913,129,639,935

Phew! This is roughly 10^{77} , which is of the same kind of order as the number of atoms in the visible universe. In other words, brute-forcing a key this long is equivalent to counting every atom in a sphere of space 100 billion light years across. How long would that take, then?

Well, you can do the math, but a *long* time, many billions of billions times longer than the current age of the Universe, even if Eve throws every

computer on Earth at the problem (which she could conceivably do, using something like a botnet). That's as close to uncrackable as makes no difference, for our purposes.

Yes, computers get faster over time, so in a hundred years it'll take much less time to crack keys than it does today. And it's possible to imagine an advanced civilisation building vast computers, the size of solar systems, dedicated solely to brute-forcing Alice's messages from long ago.

But pretty soon, the limiting factor becomes energy, rather than time. Computation takes energy, and the bigger the computer, the more energy it uses. Bruce Schneier observes (in [Applied Cryptography](#)) that the usable energy released by the average supernova is about enough to cycle a 219-bit counter through all its possible states.

To enumerate all possible 256-bit (32-byte) keys, then, would need the energy equivalent of $2^{37} \approx 137,000,000,000$ supernovae. That's roughly the number of stars in the Milky Way, so an attack like this would require burning up a decent-sized galaxy. Even a highly advanced civilisation would wince a bit at that. They might need the galaxy to live in, for example.

And packing all that energy into too small a volume would create a black hole, so the computer would have to be very large, which makes it slow, because even signals at the speed of light would take a long time to travel across it. So Galactic Emperor Eve, as powerful as she is, is still going to have a hard time cracking Alice's 256-bit key.

To put it another way, sufficiently long keys make brute-force cracking a poor choice of strategy. It would always be far cheaper for Eve just to steal the key, or to bribe someone close to Alice to steal it. (Or to drug them and hit them over the head with a \$5 wrench until they tell her the key.)

Multi-byte keys

Extending our cipher scheme to *multi-byte* keys, then, would be a long step towards the kind of ciphers we need in realistic applications. It sounds

difficult, but what's really involved? Let's have a think.

Right now, the enciphering process is something like this: for each byte of the plaintext, we add the key byte to it to yield the corresponding byte of the ciphertext.

Embiggening the key

Suppose we had a two-byte key, though. Now what would we do? For the first byte of the plaintext, we'd add the first byte of the key. For the second byte of the plaintext, we'd add the second byte of the key. That's straightforward.

And for the third byte of the plaintext, we'd go back to the first byte of the key, and so on. Essentially, we'd just keep looping around the key bytes as we move through the plaintext. That sounds doable, doesn't it?

And this scheme works just as well with a key of three bytes, or even more. So let's say that the key can now be a *slice* of bytes of any length, instead of just a single byte.

Updating the test cases

We don't actually need to change our existing tests that much. We can think of the current scheme as using a key of length 1, so all our existing 1-byte-key test cases should still pass when we've generalised the code to handle keys of any length.

First, we need to change our test case struct, because instead of a single byte for the key, we want to supply a slice of bytes. Here goes:

```
var cases = []struct {
    key      []byte
    plaintext []byte
```

```
    ciphertext []byte  
}
```

And the first test case is easily updated to suit this new type definition:

```
{  
    key:      []byte{1},  
    plaintext: []byte("HAL"),  
    ciphertext: []byte("IBM"),  
},
```

This part is worth looking at more closely:

```
key:      []byte{1},
```

The curly braces indicate that we're giving a []byte literal (in this case, a slice of just one element, the number 1). This looks very like *converting* something to a []byte, as we do with the other two fields, but that uses parentheses, not curly braces.

Let's delete the other cases for now, since this will be enough to get us started, and we can come back and fill in some more demanding test cases later.

Bytes in, bytes out

The test currently doesn't compile, so let's address that problem first:

```
cannot use tc.key (variable of type []byte) as byte value in  
argument to shift.Encipher
```

That makes sense. After all, we're *saying* that Encipher will need to take a []byte as the key now, not just a single byte.

So let's make that change to the function signature:

```
func Encipher(plaintext []byte, key []byte) (ciphertext []byte)
```

We'll see similar compile errors in the tests for `Decipher` and `Crack`, of course, but that's expected. Let's comment out those tests for now, and focus on getting `Encipher` working again.

If you wrote `Decipher` using a call to `Encipher`, this also won't compile for now, so comment out the `Decipher` function too. And since `Crack` uses `Decipher`, we'll comment that out too.

Fixing `Encipher`

Even though we've now fixed the errors in `TestEncipher`, the `Encipher` function itself still doesn't compile yet:

```
invalid operation: b + key (mismatched types byte and []byte)
```

This also makes sense, since you can only add together values of the same type. Adding `b` (the current byte of the plaintext) to `key` (a slice of bytes) wouldn't be meaningful. What do we actually want here?

We don't want to add the *whole* key to each plaintext byte. Instead, we should add each byte of the key in turn, and once we've used them all, we should start again at the first key byte.

A magical solution

That sounds a bit complicated, so let's think it through. Suppose there were some wonderfully convenient variable `magic` which would automatically select the appropriate byte of `key` for us.

Then we could write simply:

```
ciphertext[i] = b + key[magic]
```

When `i` is zero—that is, when we’re considering the first byte of the plaintext—then `magic` would also be zero. As `i` increases, so `magic` would increase also. That would be great, right?

So could `magic` be simply `i` itself? Would this work?

```
ciphertext[i] = b + key[i]
```

Unfortunately not:

```
FAIL: TestEncipher/HAL_[1]_=IBM (0.00s)
panic: runtime error: index out of range [1] with length 1
```

Constraining the key index

What’s happening here? Well, we’re enciphering the plaintext “HAL” with the key `[]byte{1}`. The first time through the loop, `i` is zero, indicating the plaintext letter H, and `key[0]` is the first element of the key, which is the byte 1. So far, so good.

But when `i` increases to 1, indicating the plaintext letter A, now we have a problem, because `key[1]` doesn’t exist. The key slice is only one element long, so that explains the “index out of range” error.

Which key byte *do* we want now? Well, zero, because that’s the only valid index of `key` there can *ever* be: there’s only one element in the slice. For longer keys this wouldn’t be true, of course, but we can still expect the key to be shorter than the plaintext for most messages.

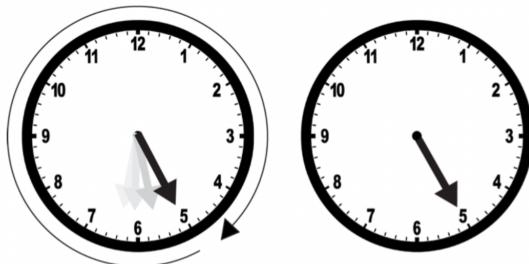
So we need a way to compute an index of `key` that increases along with `i`, but never exceeds the highest valid index of `key`, instead wrapping around back to zero.

Modulate your enthusiasm

This sounds like the mathematical operation called *modular reduction* (from “modulus”, the Latin word meaning “a small quantity”). So what’s that?

A modulo B, or just “A mod B” for short, is defined as the remainder—the *residue*—that’s left after dividing A by B as many times as you can. For example, 12 modulo 10 is 2, because 10 goes into 12 just once, leaving a remainder of 2. Similarly, 5 mod 2 is 1, because 2 goes into 5 twice, leaving remainder 1.

Modular arithmetic like this is sometimes called “clock arithmetic”, and you can see why, because it’s like adding times on an analogue clock face. What time is 12 hours later than 5 o’clock? Simply adding 5 and 12 would give us “17 o’clock”, but of course there’s no such time.



Instead, the clock “wraps around” to 5 o’clock again. To put it another way, $17 \bmod 12$ is 5.

Go has a built-in operator for this: %, the integer remainder operator. We can use it to find the residue of one number modulo another. For example:

```
fmt.Println(17%12)
// Output:
// 5
```

The modulus operator

This sounds like just what we want. Since the key index needs to increase with *i*, but be constrained to the length of the key slice, we can use the % operator like this:

```
ciphertext[i] = b + key[i%len(key)]
```

Yes, it looks a little forbidding at first, but that's only because the % operator is unfamiliar. In fact, it's used fairly often in Go for exactly this kind of thing: iterating repeatedly through the elements of a slice as some other number increases.

If you wanted to make it even clearer what's going on, you could write instead:

```
magic := i%len(key)
ciphertext[i] = b + key[magic]
```

Here's the complete `Encipher` function, then:

```
func Encipher(plaintext []byte, key []byte) (ciphertext []byte) {
    ciphertext = make([]byte, len(plaintext))
    for i, b := range plaintext {
        ciphertext[i] = b + key[i%len(key)]
    }
    return ciphertext
}
```

([Listing shift/6](#))

Refactoring will continue until morale improves

So, does this work? All signs point to yes, because `TestEncipher` now passes. Next, let's fix deciphering. Uncomment the `Decipher` and `TestDecipher` functions and we'll see what we can do.

GOAL: Make the corresponding changes to `Decipher` and `TestDecipher` so that the test passes.

HINT: `Decipher` still works in much the same way as `Encipher`, so a similar kind of change should do the job. Note that now we longer have the option of calling `Encipher` with a negative key to do the deciphering. That wouldn't make sense with multi-byte keys, so we'll need to use the loop-based version of `Decipher`.

You've got this!

SOLUTION: The change needed for `Decipher` to support long keys is more or less the same as that for `Encipher`. We'll use the `%` operator to select the current key byte as we loop, and of course we'll *subtract* it from the ciphertext byte, rather than adding.

Here's the version of `Decipher` we need, then:

```
func Decipher(ciphertext []byte, key []byte) (plaintext []byte) {
    plaintext = make([]byte, len(ciphertext))
    for i, b := range ciphertext {
        plaintext[i] = b - key[i%len(key)]
    }
    return plaintext
}
```

([Listing shift/6](#))

Testing longer keys

While our existing test cases were enough to establish that the program works for single-byte keys, we'll need to extend them a bit now, because there are new potential bugs to worry about that weren't possible before.

To show that `Decipher` behaves incorrectly with longer keys, we'd need some test cases with longer keys, and we don't have any yet.

Let's fix that.

Designing the test cases

GOAL: Add some test cases to demonstrate the correct behaviour of `Encipher` and `Decipher` with multi-byte keys.

HINT: It doesn't matter what specific cases we use, so long as they demonstrate the effect of a few different multi-byte keys on some plaintexts.

In each case, then, we can choose a slice of plaintext bytes and a slice of key bytes, and between them we can work out what the corresponding ciphertext should be.

Here's a simple example:

```
{
    key:      []byte{1, 2, 3},
    plaintext: []byte{0, 0, 0},
    ciphertext: []byte{1, 2, 3},
},
```

([Listing shift/6](#))

See what else you can come up with. If you find any cases that don't pass with your version of `Decipher`, maybe you found a bug! (Or the test case is wrong, in which case you still found a bug.)

SOLUTION: An all-zero plaintext, as in our example, is easy to understand, but not as rigorous a bug detector as we'd like.

For example, suppose that instead of properly combining the key bytes with the plaintext bytes, we just ignored the plaintext and used the key bytes directly as the ciphertext. That's wrong, but we can imagine making such a mistake.

What would happen with our suggested test case if that bug were present, then? Well, the key is "1 2 3", and the correct ciphertext is "1 2 3", so our buggy code would still pass the test, even though it's only by coincidence (because the ignored plaintext happened to be "0 0 0").

So let's add a case where that *can't* happen:

```
{
    key:      []byte{1, 2},
    plaintext: []byte{0, 1, 2},
    ciphertext: []byte{1, 3, 3},
},
```

([Listing shift/6](#))

Not only does this prove (for fairly small values of "prove") that we incorporate the plaintext bytes, it also requires that we *repeat* the key when it's shorter than the plaintext. That's another bug we could have imagined: forgetting to repeat the key. This case catches that one too.

You can add more cases, if you like, and that's usually a good idea, but I think you get the point. Providing these new cases pass, and they do, we can

feel reasonably confident about the changes.

Sharpening the tools

Good progress, but since we changed the signature and behaviour of `Encipher` and `Decipher`, we'll also need to make some updates to our command-line tools.

As a refresher, here's the existing `main` function from the `encipher` tool:

```
func main() {
    key := flag.Int("key", 1, "shift value")
    flag.Parse()
    plaintext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    ciphertext := shift.Encipher(plaintext, byte(*key))
    os.Stdout.Write(ciphertext)
}
```

([Listing shift/4](#))

We can see that, with the changes we've made to `Encipher`, this no longer compiles:

```
cannot use byte(*key) (value of type byte) as []byte value in
argument to shift.Encipher
```

When the right answer is wrong

It's the same issue as we had when updating the `Encipher` test for multi-byte keys, isn't it? Where the function expects a `[]byte` for the key, we're only

sending a single byte. So what's the fix?

We could turn it *into* a byte slice by writing a literal, like this:

```
ciphertext := shift.Encipher(plaintext, []byte{*key}))
```

Now the *compiler* is happy, but this still doesn't seem quite right to me.

GOAL: Can you figure out what's wrong?

HINT: Think about what we're really trying to do here. Go is pretty smart, but it's not smart enough to understand what our changes to the program are *about*: we're trying to expand it to use multi-byte keys.

It correctly identifies the type mismatch between `byte` and `[]byte`, but it doesn't give us any advice about how to fix it. The *wrong* way would be to convert `*key` to a `[]byte`, as in the example.

Turning a single byte into a slice of bytes would just give us a one-element slice, which is pointless. Actually, the problem isn't on this line, it's somewhere else, because the real issue is that `*key` should *already* be a slice at this point.

See if you can arrange that.

SOLUTION: With the code as it stands, we can still only ever encipher with single-byte keys... which was the very thing we came here to fix! As we saw, the problem isn't really at the point in the code where Go complains.

Instead, we need to work backwards to see why `*key` is a single byte, and fix that instead.

What kind of flag do we need?

Clearly we need Alice to be able to supply a multi-byte key if she wants to. But how to do that? `flag.Int` creates a flag that takes a single integer as argument. It would be nice if there were something like `flag.IntSlice`, allowing us to take any number of numbers, like this:

```
encipher -key 1,2,3...
```

Unfortunately there isn't. But hold up: we already know that the key, in some sense, is just a single number, no matter how bytes it takes to express it. For example, if we had a 32-byte key, we could express it as either a series of 32 integers (one for each byte), *or* as a single (probably very large) integer.

So it looks like we can stick with supplying a single value for the key after all. But we don't really want to have to deal with potentially enormous numbers *as* integers. A 32-byte (that is, 256-bit) key would be much too big to fit in a Go `int64`, for example, which can only hold 8 bytes (or 64 bits) worth of information.

The joy of hex(adecimal)

Instead, there's a neat and concise way to write large integers: as a string, using *hexadecimal* notation. This (from the Latin words for “six” and “ten”) is a way of writing numbers in base 16. So what's that?

Binary notation

Well, we're already familiar with base 10 (the plain old decimal system), and if you've ever seen *binary* numbers written down then you know what base 2 looks like:

00011001

The “base” here is another way of saying “place value”. In decimal notation, each digit counts as ten times bigger than the one to its right, but

in binary, or base 2, each place is only worth *twice* as much as its neighbour.

For binary notation, then, we don't need many possible digit *values*: only two, in fact. You can use 0 and 1, true and false, or whatever pair of symbols you like (call me a traditionalist, but I like 0s and 1s).

What about base 16, then? Well, just as there are two possible digit values in binary, and ten in decimal, in hexadecimal ("hex" for short) there are *sixteen* possible digit values. Can you guess what they all are?

Well, if you wanted to write in hexadecimal, you might start with the decimal digits we're already familiar with: 0 through 9. But we still need another six, so what could we use instead of numbers?

That's easy: we'll just borrow some letters! We can use the letters A through F for these extra digit values. So hexadecimal digits go from 0-9, and then A-F before "carrying" over to the next place.

For example, what decimal value is represented by this hexadecimal number?

3B

We can work it out by considering each place value, starting from the right. The rightmost digit represents "ones", and its value is "B" (decimal 11). Next, we find a three in the "sixteens" place, and $3 \times 16 = 48$. Add the 11 we started with, and that gives a total of 59.

So 59 in decimal would be written as "3B" in hex. And we often write it with a leading 0x, meaning hexadecimal, to avoid any confusion, thus: 0x3B.

Because we have more possible digit values available, we can write large numbers more compactly in hex than in decimal. That's just what we need for our new key flag, isn't it?

A string flag

It sounds like we can just take a string flag containing a hex value, and then interpret it as a slice of bytes representing the equivalent number. Let's try.

GOAL: Modify encipher to accept a key value from Alice as a hex string.

HINT: Once you've got the key string, the standard library's `hex.DecodeString` function will turn it into a byte slice for you (but beware, this can fail if the string is invalid).

SOLUTION: Changing key from an integer to a string flag is no problem:

```
keyHex := flag.String("key", "01",
    "key in hexadecimal (for example 'FF')")
```

And we can easily transpose that string into a `[]byte` using the `encoding/hex` package:

```
key, err := hex.DecodeString(*keyHex)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
```

As you can see from this code, there can be an error in this process: for example, if Alice enters a string that doesn't happen to be a valid hexadecimal number. There's nothing we can do about that except report it and exit, leaving Alice to try again.

Where's the BEEF?

So here's the updated program, now enhanced with the very latest in multi-byte-key functionality:

```
func main() {
    keyHex := flag.String("key", "01",
        "key in hexadecimal (for example 'FF')")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    plaintext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    ciphertext := shift.Encipher(plaintext, key)
    os.Stdout.Write(ciphertext)
}
```

([Listing shift/6](#))

Go ahead and make the same changes to the `decipher` program ([listing shift/6](#)), and we'll try out our new feature:

```
echo "Hello, world" | go run ./cmd/encipher -key DEADBEEF
```

&* [M..fM*S.%

Looks reasonable. Let's try the round-trip through `encipher` and back through `decipher` to see if we get back what we started with:

```
echo "Hello, world" | go run ./cmd/encipher -key DEADBEEF \
| go run ./cmd/decipher -key DEADBEEF
```

Hello, world

Not bad!

Changing frequencies

Yes, DEADBEEF is a valid hex number, to the amusement of programmers everywhere: it's equivalent to 3735928559 in decimal. And it corresponds to a key of 4 bytes (DE, AD, BE, EF). In fact, any given byte can be written as exactly two hex digits, which is convenient.

And you can see from the output that, unlike with our single-byte version, the same plaintext letter does *not* always produce the same ciphertext letter. For example, the “ll” of “Hello, world” is enciphered as “[M”.

This doesn't make our updated cipher completely proof against letter frequency analysis, but it does make Eve's task a lot harder—which is all we can ever hope to do, of course.

6. Cribs

The three laws of security:

1. *Absolutely secure systems do not exist.*
2. *To halve your vulnerability, you have to double your expenditure.*
3. *Cryptography is typically bypassed, not penetrated.*

—Adi Shamir, “[Cryptography: State of the Science](#)”



Okay, so we beefed up our cipher (or BEEFED it up, if you prefer) to make it much more difficult for Eve to crack by brute force. It'll still be *possible*, of course: it'll just take longer.

How much longer, though?

Cracking long keys

Let's see if we can update our `crack` function to recover multi-byte keys. Over to you again for another coding challenge!

GOAL: Update `crack` to deal with multi-byte keys.

HINT: First, let's review the previous version of crack, which worked with single-byte keys:

```
func Crack(ciphertext, crib []byte) (key byte, err error) {
    for guess := range 256 {
        result := Decipher(ciphertext[:len(crib)], byte(guess))
        if bytes.Equal(result, crib) {
            return guess, nil
        }
    }
    return 0, errors.New("no key found")
}
```

([Listing shift/5](#))

Nice and simple, in principle: we try every possible key byte in turn, seeing if it correctly deciphers the message according to the crib. If so, we return the key. If we've exhausted all the possible keys without finding one that works, we give up.

What needs to change?

We can keep the existing loop that tries every possible byte value in turn, but we'll need to add a new outer loop that repeats this guessing process for each byte of the key.

One problem here is that we don't know how *many* bytes the key contains. No matter how long the key we try, there's no way to know that adding one more byte wouldn't crack the message. So we might have to put an arbitrary limit on how long we keep guessing.

A byte at the problem

SOLUTION: Let's start with the outer loop. We'd like to write something like this:

```
for k := range KEYLENGTH {  
    // try to guess key byte k  
    for guess := range 256 {  
        ... // check guess against crib  
    }  
}
```

But already we have a problem: what is KEYLENGTH? We don't know, because we can't rely on Alice being considerate enough to tell us the length of the key she's using. It could be one byte, or two, or more. In fact, as we discussed, it could be arbitrarily long. What to do?

Well, there has to be *some* cutoff, or we'd just go on guessing forever. Since we already established that 32-byte keys are enough to keep our secrets safe for a hundred bajillion years, maybe that's a sensible limit on our search:

```
const MaxKeyLen = 32
```

In practice, Eve won't have the patience (or the longevity) to wait until the program has exhausted all the possibilities of such a long key. Later on, we'll get some idea of just how long that would take. But this is certainly a reasonable upper bound on the keys it's worth trying to crack.

So is this all we need for the terminating condition of our loop? In other words, can we write:

```
for k := range MaxKeyLen {
```

Length limitations

Well, not quite, because there's another limitation on the key length: the length of the ciphertext itself. In other words, if you have, say, a 12-byte ciphertext, then the effective key can't be more than 12 bytes long itself.

Sure, you can *use* a longer key if you like, but the extra bytes won't affect the ciphertext in any way. At least, not with our cipher scheme. So we needn't waste time trying to guess them.

In practice, most messages will probably be longer than 32 bytes, meaning that we'll run out of key bytes before we run out of ciphertext. The smaller of the two values will be our loop limit, then:

```
for k := range min(MaxKeyLen, len(ciphertext)) {
```

A bit ugly, but never mind: we're trying to save the world here, not win an "elegant code" contest. We're making progress, though: we can now loop over every byte of the key, and use our existing inner loop to try to guess what it is.

Checking our guesses

What exactly do we mean by "try to guess", though? Let's say we're considering the first byte of the key, and the first guess we try is 0. How can we check whether this is the correct value for `key[0]`?

Well, we know that `plaintext[0] + key[0] = ciphertext[0]`. We don't have `plaintext`, but we do have `crib`, which is guaranteed to replicate at least the first part of the plaintext. So suppose we did something like this:

```
for guess := range 256 {
    result := ciphertext[k] - byte(guess)
```

```
    if result == crib[k] {
        key = append(key, byte(guess))
        break
    }
}
```

We try our guess for this key byte by subtracting it from the corresponding byte of the ciphertext. If the result is the same as the corresponding byte of the crib, then our guess is correct, and we can add this byte to the key we've built up so far.

Knowing when to stop

That's terrific, and we now know how to check each guess for each byte in the key. But there's one more thing missing: how do we know when we're *done*?

In other words, suppose the key is 2 bytes long, and we've correctly computed both bytes. What stops us continuing round the outer loop, trying to guess a further 30 useless bytes?

Well, just as with the previous version of `crack`, we'll know we've found the whole key when we can use it to decipher the whole message (or at least that part of it corresponding to the crib):

```
if bytes.Equal(crib, Decipher(ciphertext[:len(crib)], key)) {
    return key, nil
}
```

But, as we saw before, this may *never* happen, however good our code, so we also need a way to report that we failed honourably:

```
    return nil, errors.New("no key found")
```

A proper little cracker

So here's the complete multi-byte version of crack:

```
const MaxKeyLen = 32

func Crack(ciphertext, crib []byte) (key []byte, err error) {
    for k := range min(MaxKeyLen, len(ciphertext)) {
        for guess := range 256 {
            result := ciphertext[k] - byte(guess)
            if result == crib[k] {
                key = append(key, byte(guess))
                break
            }
        }
        if bytes.Equal(crib, Decipher(ciphertext[:len(crib)],
key)) {
            return key, nil
        }
    }
    return nil, errors.New("no key found")
}
```

([Listing shift/6](#))

This certainly isn't the only possible way we could write it, and I'm sure the version you came up with is a little different. But as long as it passes the tests, it is by definition correct, and that's what we care about.

Speaking of the tests, we can now uncomment them and run them to see what's what:

```
invalid operation: tc.key != got (slice can only be compared to nil)
```

That's fair. Remember, after `crack` returns its result, the test checks it against the input key to see whether it got the right answer:

```
if tc.key != got {
```

But these values are now both slices, not bytes, and as Go points out, one does not simply compare slices with `!=`.

We could use the standard library function `slices.Equal`, which works for any kind of slice, but as it happens we can be even more specific in this case. We know our result is a slice of bytes, so we can compare it using `bytes.Equal`:

```
if !bytes.Equal(tc.key, got) {
```

So here's the updated test in full:

```
func TestCrack(t *testing.T) {
    t.Parallel()
    for _, tc := range cases {
        name := fmt.Sprintf("%s + %d = %s", tc.plaintext, tc.key,
                            tc.ciphertext)
        t.Run(name, func(t *testing.T) {
            got, err := shift.Crack(tc.ciphertext,
                                   tc.plaintext[:3])
            if err != nil {
                t.Fatal(err)
            }
        })
    }
}
```

```
    if !bytes.Equal(tc.key, got) {
        t.Fatalf("want %d, got %d", tc.key, got)
    }
}
}
```

([Listing shift/6](#))

What about the old test we had that checks `crack` returns an error when it can't find the key? Should we update it too? Well, it's not really meaningful anymore. If you think about it, given any ciphertext and any crib, there will always be *some* key that transforms the ciphertext into the crib. We don't really need the test anymore, so let's delete it.

In general, it's a good idea to get rid of any tests that are no longer providing value. All code is technical debt, and tests are no exception. Keep them lean and focused.

Updating the `crack` tool

Now that we can crack longer keys, the next step is to add that new functionality to our command-line tool.

Applying brute force

The only change needed, in fact, is that we no longer have to cast `key` to a byte when passing it to `Decipher`, because it's already the type we want:

```
func main() {
    crib := flag.String("crib", "", "crib text")
    flag.Parse()
    if *crib == "" {
```

```

        fmt.Fprintln(os.Stderr,
                     "Please specify a crib text with -crib")
        os.Exit(1)
    }
    ciphertext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    key, err := shift.Crack(ciphertext, []byte(*crib))
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    plaintext := shift.Decipher(ciphertext, key)
    os.Stdout.Write(plaintext)
}

```

([Listing shift/6](#))

Crib considerations

So let's try it out!

```

go run ./cmd/encipher -key DEADBEEF <tiger.txt >enciphered.bin
go run ./cmd/crack -crib 'The tiger' <enciphered.bin

```

The tiger appears at its own pleasure. When we become very silent
 ...

Not bad. We still managed to brute-force the key, even though there are now over 4 billion possibilities, instead of just 256. And it was still pretty fast!

We didn't need a very long crib, either: just 2% of the length of the message, which seems quite achievable.

Could we still have cracked this key, even with a shorter crib? Let's try:

```
go run ./cmd/crack -crib 'The' <enciphered.bin
```

```
The1.XG...r.pea..A.@8.. owQ_L..B..e. h.TNb.41.eco~.V..H1.ile.A.@
...
```

Um. That doesn't look right (apart from the first three letters). So what went wrong?

A false deciphering

Actually, thinking it through, you can see what happened, can't you? We tried to guess the first byte of the key; eventually we tried the byte `DE`, and it worked. We recovered the first letter of the crib: "T".

Next, we worked our way round to correctly guessing that the second byte of the key is `AD`, and used it to recover the second crib letter: "h". The third byte of the key is `BE`, and it yields the crib letter "e", which is right.

But, according to the code, once we've correctly deciphered the whole crib, we consider the key found, and return it. However, the key `DE AD BE` is a byte too short. Cycling repeatedly through this key as we decipher the message gives the wrong result, as we just saw.

On being the right size

I don't think it's obvious at first glance, but we can now see why a useful crib needs to be *at least as long as the key*, with this cipher scheme.

With a crib that's shorter than the key, we can only recover as much of the plaintext as we already have in the crib, which is no use. (It does get us *part*

of the key, which is better than nothing. It at least reduces the remaining search space.)

If this theory is right, then, a crib *exactly* as long as the key should still be sufficient for a correct deciphering. We're using a four-byte key (DE AD BE EF), so the string "The " (including the space) should be enough.

Let's try:

```
go run ./cmd/crack -crib 'The ' <enciphered.bin
```

The tiger appears at its own pleasure. When we become very silent
...

Good. Story checks out. Let's move on.

7. Passwords

MICKEY: *It's asking for the password.*

THE DOCTOR: "Buffalo". Two *Fs*, one *L*.

JACKIE: *So, what's that website?*

MICKEY: *All the secret information known to mankind.*

—Doctor Who, "["World War Three"](#)"



We improved our very simple cipher scheme quite a bit, just by adapting it to work with longer keys. And we've seen why longer keys make the system more secure: if all Eve can do is try to guess the key, a longer key makes that job harder.

Security

But do all keys of a given length make an enciphered message equally secure? Surely not. For example, a key of all zeroes, in our cipher system, doesn't alter the plaintext at all. So, while size *does* matter, there are other aspects of choosing a secure key that we need to think about.

So, what does it mean for a key to be more or less secure, and how can we choose or generate secure keys for use with our cipher?

Password spaces

Let's think about *passwords*, for example, to bring this problem closer to most people's everyday experience. A password is a kind of cryptographic key, but there's no message (or the message is simply "It's me!"). Instead, being in possession of the correct password gives you access to some *resource*, like a computer or a bank account, rather than a message.

But the same kind of rules that govern keys in general apply to passwords, too: to be secure, they should be hard to guess, for example. In our discussion of keyspaces, we saw that making the number of *possible* keys larger automatically makes the key harder for Eve to guess.

That's because enumerating the possible keys requires computation, and that takes some irreducible amount of energy. Eve can guess keys faster by using more computers in parallel, but her final electricity bill will be the same, regardless.

But we've also implicitly assumed that Alice is completely *unbiased* in her choice of key. In other words, as Bruce Schneier puts it:

If the key is 64 bits long, every possible 64-bit key must be equally likely.

—“[Applied Cryptography](#)”

Dictionary words

But a password is a special kind of key in another way: traditionally, it was expected that a person would be able to *memorise* their password. Human brains, though, impressive as they are in some ways, aren't all that good at remembering arbitrary combinations of letters and symbols (or, for that matter, 64 randomly-generated bits).

So when people are allowed to choose their own passwords, they tend to pick English words (or whatever their favourite language is). This is already a bad idea on the grounds of key length, because most words are pretty short.

The average English word is about five letters long, and as we've already seen, that's well within the capacity of current computers to guess in a few seconds, simply by enumerating all possible combinations of letters. A fourteen-letter password would take much longer to brute-force, perhaps 100 million years, but such words are relatively uncommon in English (when's the last time you managed to work the word "abdominoplasty" into conversation?)

It gets worse, though, because if Eve knows that Alice's password is an English word, she can take a short-cut. Rather than enumerate all possible combinations of letters, she can start by trying every word in the dictionary, because there are far, far fewer of those. In other words, the set of combinations of N letters that happen to form English words is a tiny subset of all *possible* combinations of N letters.

Extremely guessable passwords

Only if the dictionary search fails does Eve need to fall back to brute-forcing. But it's very likely that Alice's password *will* be found by a simple lookup. In one study cited by Schneier, such a *dictionary attack* succeeded in finding passwords for 40% of the accounts targeted. In other words, at least 40% of people are picking extremely guessable passwords.

Indeed, just 25 or so common words make up around 10% of all passwords, including `password`, `qwerty`, `i loveyou`, `admin`, `lovely`, `welcome`, and variations on `12345`, `123456`, `1234567`, and so on. (If you want to take a minute to go and change some of your passwords right now, go ahead. I'll wait.)

So while Eve *could* explore the keyspace sequentially, as our key-cracking tool does, or randomly, she'd be much better advised to start with the widely-available [common password list](#), followed by the English dictionary.

Next on Eve's list should be simple *obfuscations* of common passwords, such as those produced by substituting 1 for i, 0 for o, and so on. Then she might move on to pairs of words, including obfuscated words, joined by symbols (s3cr3t-passw0rd), then combinations of three words, and so on.

Although there are lots of English words (my system dictionary reports about a quarter of a million), and even more word *pairs* (60 billion), that's still just peanuts compared to the total keyspace. So it absolutely makes sense for Eve to try these possibilities first, if she knows that Alice has a memorisable password.

Passphrases

Obfuscated dictionary words, then, are both easy for Eve to guess and surprisingly hard for Alice to remember accurately (was it secr3t or s3cret?) Instead, when security is paramount, Alice should use some well-tested computer program to generate a long, random password from as wide a range of characters as possible. She won't be able to remember it, but that's what password managers are for. Use them.

But what if it's something that Alice just *has* to be able to memorise? For example, in a situation where she won't be able to use password manager software or some other kind of cryptographic vault. Can a memorisable password be secure?

Well, there's a trick for this: instead of using just one word, use *lots* of words. For example, "Colourless green ideas sleep furiously". That's 38 characters including spaces (37 if you're American), which is long enough to defeat even the most patient brute-force attacker. Yet, it sticks in the mind. I'd be prepared to bet you'll remember it even after you've finished reading this book (so don't actually use that example).

Long phrases like this aren't entirely proof against dictionary attacks, but they do make such attacks much, much harder and more expensive. For systems that limit the length of passwords, you can use the initial letters of some long phrase instead:

Shall I compare thee to a summer's day?
Thou art more lovely and more temperate.

becomes:

S!ctasd?Tamlamt

Rules

Just out of interest, though, how big *is* the keyspace for a password of a given length, assuming that Alice is smart enough not to choose dictionary words, but instead picks characters at random in some way?

Character sets

That depends on the set of characters from which Alice is allowed to choose. Let's take a five-character password, for example, just because it makes the numbers easier to deal with.

If Alice is restricted to only uppercase letters, and only those from the English alphabet, then there are 26 possibilities for the first letter, 26 for the second, and so on. That gives us $26^5 \approx 12,000,000$ possible passwords.

Fine, so how does that change if we add lowercase letters to the set? Now there are 52 choices for each character, giving $52^5 \approx 380,000,000$. Quite a difference: the keyspace (and, equivalently, the average brute-force time) is 32 times bigger. We doubled the keyspace for each character, giving us $2^5 = 32$ times as many possible 5-character passwords.

Let's add digits and symbols to the list: in fact, let's add every character it's possible to type on a computer keyboard, of which there are about 100 in total. So that's $100^5 = 10,000,000,000$. That's about a thousand times bigger keyspace than with just the uppercase letters.

Of course, we know that making the password *longer* would also increase the keyspace, exponentially with the number of characters, but let's ignore

that for now and focus only on the effect of enlarging or reducing the set of *allowed* characters.

Password policies

You've probably used websites or applications that enforce certain rules about the passwords you can choose. For example, a typical rule might be that the password must include at least one digit.

So what effect does that have on the keyspace, and thus the security of the password? Let's run the numbers.

As we've seen, the keyspace of a 5-character password using all the typeable symbols is about ten billion. But if Eve knows that one of those characters *has* to be a digit, that makes her search much easier. Now she only has to search the keyspace of a *four*-character password, and for each guess, try it with each of the ten possible digits in each of the five places it could be. That's fifty guesses for each four-character password. So the keyspace is reduced to $50 \times 100^4 = 5,000,000,000$, or half the size it was with an unrestricted password.

What the hey? It seems like adding that extra "security rule" made the password twice as easy to guess, and indeed that's the case. Let's add another rule: the password must include at least one uppercase letter. Sadly, that reduces its security still further, for the same reason, and so does every extra restriction we place on Alice's choice of characters.

We'd better stop adding password rules quickly, before we leave Alice with only a handful of possible passwords, and Eve with a very easy task of brute-forcing them! We just learned something very useful: password policies generally make the security of the system worse, not better.

You're really not helping

So why do application designers impose such restrictions in the first place? Presumably they *mean* well, and their argument for requiring things like digits and symbols might be that it prevents users from choosing simple

dictionary words. Or, at least, it forces them to add a digit and a symbol to their chosen word. That's not a terrible idea in principle.

But by helping users who are bad at choosing passwords, these policies actively harm users who are good at it. That seems a shame. If you're writing software that lets users choose passwords, please don't add restrictions of this kind. If your boss insists on you adding such a feature, show them this book (better still, get them to buy their own copy).

Don't get me wrong: helping users pick strong passwords is a good thing to do. This just isn't the right way to do it. Wouldn't it make more sense, indeed, just to look up the user's proposed password in the dictionary, and if it's there, reject it? Some websites do show a "password strength" indicator as you type, which is nice, but who knows how it works or what it measures?

The simplest and most effective thing you could do to improve the security of your users' passwords is probably just to require a minimum length of 14 characters. This means they will almost certainly have to use a password manager, so please don't also disable their ability to paste into the password field.

While we're on the subject of irritating, misguided, and counterproductive password policies, some applications or even operating systems require users to regularly change passwords. This is a mistake. If your passwords are strong, then they won't need changing. If they're weak, then changing them won't help.

The only effect a policy like this has is to force users to write their passwords on a note stuck to their monitor, and at that point you may as well not bother with a password at all.

Maximising the keyspace

Back to cryptography. The cryptographic reason that password character restrictions are bad, as we've seen, is that every restriction on what

characters Alice can include in her password is also information for Eve about what characters it *will* include.

Quantitatively, the keyspace of a password of N characters, each chosen from a set of K possibilities, is:

$$K^N$$

Making N small (short passwords) is a bad idea, as we already know, but now it's clear that reducing K is also a mistake. If you halve the size of K , for example, the keyspace is reduced by a factor of 2^N , and two to the power of *anything* gets big fast.

Some policies are helpful, of course: imposing a minimum length on passwords, for example. Yes, this reduces the keyspace in one way, since Eve no longer has to bother trying passwords shorter than the minimum, but there aren't many of those, so it doesn't save her much labour.

Indeed, a “long password” policy increases the keyspace much more in another way, since it prevents Alice from choosing pathologically small N . The net effect is good for Alice, thus bad for Eve.

Unicode

So if reducing the possible choices for each character of a password is a bad idea, it follows that *increasing* them must be a good one. If we don't restrict Alice to choosing merely from the 100 or so typeable characters, and instead allow her to pick any *Unicode* character, how does that affect the security of the resulting password?

You know how to answer this question now: by asking another question, namely “How many Unicode characters are there?” The answer is about 150,000. So a 5-character Unicode password would have a keyspace of $150,000^5 \approx 8 \times 10^{25}$, vastly larger than a password restricted to only the typeable characters.

Sadly, though, the support for Unicode characters in many software systems is buggy or non-existent. You might even find that a system accepts your

Unicode password, but then refuses to recognise it when you enter it later. Programmers (especially non-Go programmers) often assume that all text will be limited to the 128 ASCII characters, and when it isn't, their programs usually don't work. (Try entering a poop emoji in any text field if you want to see which programs these are.)

A much more reliable way for Alice to make her password harder to guess, then, is simply to make it as long as possible. Bad programmers can still defeat her, for example by putting an *upper* limit on password length, which is crazy, but many applications have just such a rule. (Is all this bad password advice coming from Eve's website? One wonders.)

They can also compromise her security by storing and matching only the first few characters of the password (many versions of Unix still do this even today). This is even worse, because it *looks* like you have a long password, but you don't.

Generation

Assuming the system doesn't mangle or truncate Alice's password, though, and that she's not naïve enough to choose an easy dictionary word, what should she do to generate the most secure possible password of a given length?

The answer, of course, is that she should generate it *randomly*, but what does that really mean? And how should she do it?

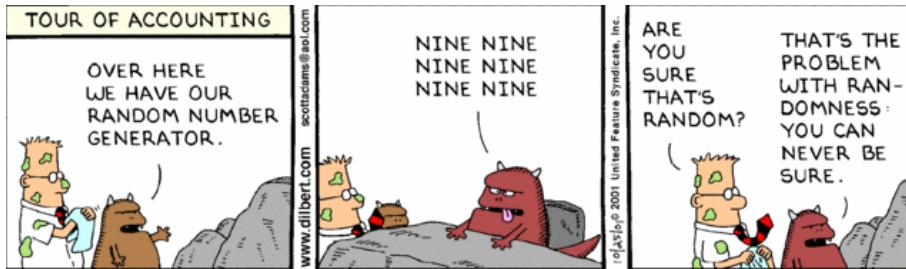
What does “random” mean?

No doubt we already have some intuitive understanding of what “random” means, but let's make it a bit more rigorous. For example, suppose I presented you with the following program as a random number generator:

```
func random() int {  
    return 7
```

}

You probably wouldn't be too impressed, but why not? I mean, 7 seems as random a number as any other. There's nothing special about it. If you hadn't seen the source code, you'd have no way of knowing in advance that the program would produce the number 7. So, if it *isn't* random, why not?



One way to answer this is to run the program a few times, and note that it always produces the same number. The problem, in other words, is that the sequence of values it produces is extremely *predictable*.

Predictability

And the same objection applies to other sequences of numbers that, taken individually, seem pretty random, but *in sequence*, reveal their underlying regularity. See if you can guess the next number in each of these sequences, for example:

17, 86, 2, 17, 86, 2, 17...

1, 2, 4, 8, 16, 32, 64...

2, 3, 5, 7, 11, 13, 17...

0, 1, 1, 2, 3, 5, 8...

Well, if you can guess it, so can Eve. In a way, programs that produce these sequences are just as predictable—just as un-random—as the one that always produces the number 7.

So that's one important aspect of randomness: it applies to *sequences*, not just individual numbers, and it requires that the sequence not be obviously predictable.

But there seems to be a fundamental problem here. Computer programs are generally *deterministic*: that is, given the same inputs, they will produce the same outputs. How could we write a program that will produce a *non-deterministic*—that is, non-predictable—result?

Binary choices

Well, we might start by asking how a human could do the same thing. To generate a secure password, for example, Alice might toss a coin and use the result to help her choose a letter. Each coin toss could reduce the number of choices by half. For example, the first toss might select which half of the alphabet she should pick from. The next will reduce that set of characters by half again, and so on.

Alice will need to make at most 5 coin tosses to pick each letter, because a sequence of 5 tosses has $2^5 = 32$ possible outcomes, more than enough to cover the 26 letters of the alphabet. If Alice is using a larger character set, then it will take more tosses to pick each character, but that's a good thing, as we've seen: the bigger the character space, the bigger the keyspace.

Note that even if Eve knows that Alice is using this password-generating scheme, that knowledge by itself doesn't help her learn anything about what Alice's key actually *is*. Unless she can spy on Alice closely enough to see the outcome of each coin toss, she still has no idea of what letters have been picked. And if she has *that* level of access to Alice's private activities, she probably doesn't need to intercept Alice's messages in any case.

Alternatively, if Alice wants to generate a cipher key, such as the 32-byte keys we've used in the shift cipher, she can use each coin toss to decide a single bit in the key. 32 bytes is 256 bits, so 256 coin tosses will produce a suitably random key.

Information

Interestingly, Alice can use the same coin-toss scheme no matter what kind of information she needs to generate. If she has to choose a random day of

the week, or one of the seven colours of the rainbow, for example, then she can do so by making just three coin tosses, because $2^3 = 8$.

It's clear from this that any kind of information can, in principle, be represented using a certain number of *bits* (binary choices equivalent to coin tosses). In fact, this is a useful way of quantifying the *amount* of information present in some set of data.

How much information is there in the binary number 1011, for example? Well, four bits. That's too easy. What about the word *go*? Well, assuming only uppercase letters are allowed, there are $26^2 = 676$ possible combinations of two letters. The information content of the word *go* is precisely the information as to *which* of those combinations has been selected.

To identify that combination uniquely, we need to represent any number up to 675, and $2^{10} = 1024$, so 10 binary bits would be enough to give each two-letter word a unique identifier (with plenty of unused identifiers left over).

We can express the same idea by saying that *go*, or any other combination of two letters, contains slightly less than 10 bits of *information*. You can write it down using more bits than that if you choose, of course, but the point is that these superfluous bits don't make the keyspace any bigger.

Knowledge

Not all information is random information, as we've already seen. For example, if Alice picks a dictionary word as her password, it doesn't matter how long the word is: Eve can easily guess it. The subset of words that are in the dictionary is small, and Eve only needs to locate Alice's password within this small set, not the larger one of all possible letter combinations.

Similarly, if Alice picks a cipher key like this:

1111111111111111

it may look like it contains 16 bits of information, but that depends who's reading it. If it's Eve, and she happens to know that Alice always picks keys consisting of either all 0s or all 1s, then there are only two possible keys that Alice *could* have chosen. So the key itself only conveys one bit of information to Eve: namely, which of the two possibilities it is.

Alternatively, if Eve knows that Alice always picks keys that are all 1s, then the space of available keys from Eve's point of view contains only one possibility. In that case, the key contains no information at all!

We'll return to the topic of randomness in later chapters, and we'll talk a little about how to measure the exact amount of *entropy*—of unknown information—in a given key, and about what sources of entropy Alice could use to generate good keys.

For now, let's return to our cipher scheme, and think about a more flexible way to operate it, using *blocks*.

8. Blocks

The security of a symmetric cryptosystem is a function of two things: the strength of the algorithm and the length of the key. The former is more important, but the latter is easier to demonstrate.

—Bruce Schneier, “[Applied Cryptography](#)”



(photo by [Mariana Bisti](#))

Great. We now have a cipher system that works with keys of any length, and we can crack those keys, given a suitable crib. We've come a long way from zero to cryptography, but there are still a few missing pieces we need before we're ready to play.

Block ciphers

First, let's distinguish between two main kinds of ciphers: *block* ciphers and *stream* ciphers. A stream cipher is very simple in principle: it operates on data one bit at a time—one binary bit, that is. In other words, given a particular bit of the plaintext, and a particular bit of the key, a stream cipher will tell you the corresponding bit of the ciphertext. No context is needed.

This makes stream ciphers highly efficient, but for various reasons it's often more convenient to use a block cipher instead. A block cipher, by contrast, enciphers or deciphers *more* than one bit at a time. Any number of bits, in fact. It deals with the plaintext (or ciphertext) in chunks, or “blocks”, of some arbitrary size.

Block ciphers operate on data with a fixed transformation on large blocks of plaintext data; stream ciphers operate with a time-varying transformation on individual plaintext digits.

—R.A. Rueppel, in G.J. Simmons (ed.), [“Contemporary Cryptology”](#)

This makes stream ciphers suitable for permanent or long-running encrypted circuits, such as telecommunications links, and for implementation in hardware, since the hardware usually “sees” a single bit at a time.

On the other hand, block ciphers are ideal for implementation in software, and for handling messages of arbitrary size.

So what kind of cipher is the shift cipher we've developed? It's somewhat like a stream cipher, because it handles data a byte at a time. On the other hand, in principle it could be regarded as a block cipher where the block size is one byte.

However, that's only in principle, because in practice the code we have always enciphers or deciphers the whole plaintext (or ciphertext) at once. Effectively, therefore, the block size is the same as the message size. And, while that's simple to implement, it's not ideal.

Why blocks?

Perhaps you can already see why: if we can't deal with the data in smallish chunks, then we have to store all of it in memory at the same time.

For small messages, that might be okay, but for larger messages, it could be a problem. Suppose we—or Alice—wanted to encipher a 10GiB file, for example. We might read that data into memory using code like this:

```
plaintext, err := io.ReadAll(os.Stdin)
```

Unfortunately, if we don't happen to have 10GiB of free memory available, and we probably don't, this code will panic. That's because `io.ReadAll` does exactly what it says: it reads *all* the data, until it sees the end-of-file (EOF) marker. Only then would we even start enciphering the data, but we'll have run out of memory well before this point.

This is silly, isn't it? We don't *need* all the data to be resident in memory just so that we can encrypt a single byte of it. Indeed, if we *had* only one byte of writable memory, so that we were restricted to reading and enciphering just one byte at a time, that would actually be fine! (It wouldn't be very efficient, but never mind.)

Since a block cipher lets us work on the data in chunks of any size we choose, then, it would make sense to use slightly larger blocks than a single byte. 32 bytes would be fine, for example, and many ciphers use blocks of about this size.

The `cipher.Block` interface

So if we want to adapt our existing code to create a practical block cipher, we don't need to change the cipher *scheme*, as such. We just need to make the cipher able to operate on data that comes in fixed-size blocks.

And there's a special interface for that, defined in the standard library's `crypto/cipher` package:

```
type Block interface {
    BlockSize() int
    Encrypt(dst, src []byte)
```

```
    Decrypt(dst, src []byte)  
}
```

([crypto/cipher](#))

If you’re not familiar with interfaces in Go, don’t worry. You can think of an interface as defining a kind of “slot” or socket that different-shaped software components can plug into. Anything that “fits” this interface can be used in the way that the interface defines.

For example, we could design any kind of cipher we wanted, and as long as we provide an adapter for it that matches the `cipher.Block` interface, anyone can use our cipher with a program that supports that interface.

Plugs and sockets

In Go, an interface is defined as a set of *methods* that the matching object must have. In this case, for some type to be a `cipher.Block`, it must implement these three methods:

```
BlockSize() int  
Encrypt(dst, src []byte)  
Decrypt(dst, src []byte)
```

So suppose we created some struct type (`shiftCipher`, let’s say), and we gave it at least these three methods. That would be sufficient to *satisfy* the interface: that is, any function that takes a `cipher.Block` as a parameter would accept an instance of `shiftCipher`.

Standard library interfaces like this are a terrific idea: think of `io.Reader` and `io.Writer`, for example. They give a big boost to productivity by allowing us to plug together many different bits of code using standardised “connectors”. And `cipher.Block` is another such connector.

Adapting the shift cipher

What do we need to change to make our existing code fit into a `cipher.Block`-shaped slot, then?

The `Encrypt` method

Well, let's take our existing `Encipher` function, for example. Here's its signature:

```
func Encipher(plaintext []byte, key []byte) (ciphertext []byte)
```

It takes the plaintext and the key, as byte slices, and returns the ciphertext. This isn't a million miles away from the signature of the `Encrypt` method in `cipher.Block`:

```
Encrypt(dst, src []byte)
```

It takes the plaintext, called `src` here, but of course you can call it what you like; only its type matters for the purposes of the interface. It also takes another byte slice `dst`, intended to hold the ciphertext, once it's been produced.

So that's one interesting change: instead of *returning* the ciphertext, our new method will need to accept a byte slice and populate it with the ciphertext as it's generated. We can arrange that, I dare say.

Where's the key?

Here's something else interesting about the signature of the `Encrypt` method: it doesn't take a `key` parameter. So how does `Encrypt` "know" what key to use when enciphering `src` into `dst`, then?

If it can't be passed in as a parameter, then we must be able to get the key from somewhere else. One clue is that, since we want to implement an interface, we're necessarily writing methods on some *type*. That means our Encrypt method will have a signature like this:

```
func (c shiftCipher) Encrypt(dst, src []byte)
```

In other words, when Encrypt is called, it will be called *on* some value of shiftCipher, and that value will be available within the method as the *receiver* variable c. Maybe that's a good place to put the key: in one of the fields on the shiftCipher struct.

Fixing the key size

This sounds promising. Now, what type should we declare for the key field? We could use []byte, of course, as we did before, and then the key can be as large or small as required. However, we're starting to get a sense of how much short keys weaken our encryption, so we'd like to enforce a minimum key length.

We've also seen that a 32-byte key is long enough for all practical purposes, so we needn't worry about handling longer keys than this. Let's make an executive decision, then, that the key will always be a fixed size: 32 bytes.

For simplicity, let's also say that we'll use a block size of 32 bytes, too. So we can write at least the following:

```
const BlockSize = 32

type shiftCipher struct {
    key [BlockSize]byte
}
```

([Listing shift/7](#))

Not bad! There's more to do, of course, but this will get us started.

What if Alice wants to use our program to encipher messages using really short keys, though? Won't our minimum key length restriction be annoying? Well, perhaps, but the annoyance is probably worth it.

We *could* accept short keys, and "lengthen" them transparently to 32 bytes to suit the API of our `shift` package. For example, we could compute the [SHA-256](#) hash sum of the key, which works out nicely at 256 bits = 32 bytes. We'll talk more about hashing later on in the book, but for now just treat it as a way of writing down a short key using an unnecessarily long string of bits.

But there's no such thing as a free lunch, and the resulting key wouldn't have any more *entropy*—any more difficulty of guessing—than the original. Suppose Alice chose a five-letter word as the key, and we expanded it cryptographically to 32 bytes. Well, all Eve would have to do is list all possible arrangements of five letters, expand them using the same hash algorithm, and try the resulting keys.

The keyspace is the same size either way. It's like writing down your password in huge letters: it's not cryptographically stronger, it just takes up more space on your monitor.

In the same way, key expansion makes it *look* like we're using a longer key, but we're really not. So let's show Alice some tough love, and require her to supply a full 256-bit key as input. We'll see some possible ways to generate such keys securely later in this book.

A cipher constructor

Since we have a new type `shiftCipher`—the block cipher—we will also need a *constructor*, so users can create instances of it. `NewCipher` sounds like a reasonable name, doesn't it? Over to you once more.

The NewCipher function

GOAL: Write NewCipher, guided by tests. It should take the key, and return an error if the key is not exactly BlockSize bytes in length.

HINT: Since the word “if” occurs in the goal sentence, we already know we need at least two tests. There are two possible behaviours: either the key is the right size, or it isn’t.

In either case, the result will be signalled by the `error` value returned by `NewCipher`. So both your tests should check this value, and fail if it’s not what’s expected for the test input.

Can you see what to do?

If the key fits

SOLUTION: Let’s start with the “right size key” behaviour, since that’s slightly easier to test.

```
func TestNewCipher_GivesNoErrorForValidKey(t *testing.T) {
    t.Parallel()
    _, err := shift.NewCipher(make([]byte, shift.BlockSize))
    if err != nil {
        t.Fatalf("want no error, got %v", err)
    }
}
```

([Listing shift/7](#))

It’s evident from this code that `NewCipher` returns *two* things, and it seems logical that the first one would be the `shiftcipher` object itself. We don’t

need it for this particular test, however, so we just ignore it using the blank identifier `_`.

The test fails if `NewCipher` returns a non-nil error, since we're saying that the key *must* be exactly `BlockSize` long, and we've ensured that by using `make` to construct it.

The actual key we're passing is all zeroes, which is weak—the weakest key in our system—but that's okay. We won't actually be using it for encryption. In fact, we don't even test that the returned cipher object “remembers” the key we passed in.

But, since we'll be writing tests for enciphering and deciphering using this object later on, and *those* tests will certainly catch that problem, we can ignore it for now.

A polite refusal

We also need to test the “wrong size key” behaviour, so let's do that. We *could* write, for example:

```
func TestNewCipher_GivesErrorForInvalidKey(t *testing.T) {
    t.Parallel()
    _, err := shift.NewCipher([]byte{})
    if err == nil {
        t.Errorf("want error, got nil")
    }
}
```

This is fine, and it's what we asked for. An empty byte slice literal `([]byte{})` has length zero, so it certainly can't be the required length. `NewCipher` should return some non-nil error, then, and this test asserts that it does.

A special error value

Can we do a little more, though? For example, it might be useful to be able to determine (in the calling code) that this specific error was caused by an incorrect key size. How could we do that?

One way would be to create a *sentinel error*: just some fixed error value that we can return in this case.

```
var ErrKeySize = errors.New("shift: invalid key size")
```

([Listing shift/7](#))

Now that this error value has a *name*, we could compare the result of `NewCipher` against it:

```
if err == shift.ErrKeySize {  
    // we know it's a key size error  
}
```

This is okay, too. But, again, could we do a little more? Ideally, we'd include some information about the specific key size that caused the problem, because this could help a programmer track down exactly why the wrong key is being used. A message like this would be great:

```
shift: invalid key size 31 (must be 32)
```

On the other hand, now we can't define a fixed error message in advance, because we don't know what the actual invalid size will be until the program runs.

Wrapping errors

One neat feature of Go that we can use here is *error wrapping*. We can take the sentinel (fixed) value `ErrKeySize`, and “wrap” it in some extra information only available at runtime. To do this, we use `fmt.Errorf` to construct the error, and the special `%w` “format verb” that wraps errors. We’ll see how to do this when we come to write `NewCipher` in a moment.

First, let’s finish the test, by checking not only that `err` is not `nil`, but specifically that it’s a wrapped instance of `ErrKeySize`, using the `errors.Is` function:

```
func TestNewCipher_GivesErrKeySizeForInvalidKey(t *testing.T) {
    t.Parallel()
    _, err := shift.NewCipher([]byte{})
    if !errors.Is(err, shift.ErrKeySize) {
        t.Errorf("want ErrKeySize, got %v", err)
    }
}
```

([Listing shift/7](#))

Note that this test fails if `err` is `nil`, which we had before, but also if it’s any error other than `ErrKeySize`.

Testing for wrapped sentinel errors using `errors.Is` is much more robust than, for example, inspecting the string value of the error. We wouldn’t want tests to start failing just because we tweaked the text of an error message slightly.

Writing NewCipher

Over to you again to solve the next problem:

GOAL: Write `NewCipher`.

HINT: The signature of `NewCipher` is effectively already decided for us by the tests:

```
func NewCipher(key []byte) (cipher.Block, error)
```

And we know that if the key is the right length, we should return a `shiftCipher` object configured with that key. If it's the wrong length, we can return a helpful error message that wraps `ErrKeySize` using `fmt.Errorf` and the `%w` verb.

SOLUTION: Here's a version that should work:

```
func NewCipher(key []byte) (cipher.Block, error) {
    if len(key) != BlockSize {
        return nil, fmt.Errorf("%w %d (must be %d)", ErrKeySize,
            len(key), BlockSize)
    }
    return &shiftCipher{
        key: [BlockSize]byte(key),
    }, nil
}
```

([Listing shift/7](#))

We take the key and check its length against `BlockSize`, returning the wrapped error if they're not equal (note the `%w`, for “wrap”, in the `fmt.Errorf` format string). But assuming the key length is okay, we create a `shiftCipher` struct, initialised with a 32-byte array containing the key, and return a pointer to it.

Implementing the `cipher.Block` interface

Note the signature of `NewCipher`:

```
func NewCipher(key []byte) (cipher.Block, error)
```

What's interesting here is that we declare the result type as `cipher.Block`—an interface type—rather than `*shiftCipher`, which is the concrete type we actually return. That tells readers that we intend this object to implement `cipher.Block`, which is a useful signal.

It also ensures that, if we should make some mistaken code change that means the `shiftCipher` type no longer satisfies `cipher.Block`, Go will catch it right away.

Testing Encrypt

So, are we done? Not quite, because we still need to implement those methods. Let's start with `Encrypt`:

GOAL: Write a test for the `Encrypt` method on `shiftCipher`.

HINT: We can't use the existing test cases any more, since they don't use 32-byte keys. It'll be annoying to write out such long keys for every test case, so you may like to create a single key that we can re-use for all the cases.

Of course, *messages* don't have to be at least 32 bytes long: if they're shorter, enciphering them will just use as many bytes of the key as necessary. So you can write some test cases about fairly short messages, which also makes the code easier to read and understand.

Since we're using the same key for every case, we can also use the same `shiftCipher` object constructed with that key. When the test loops over the cases, all it needs to do is call the cipher's `Encrypt` method and check the result.

Circumstances alter cases

SOLUTION: I think we agree that it would be a little irksome to include a different 32-byte key in each test case. Instead, let's define a single test key once and for all, and we won't even bother to write that out in full. Instead, we'll just generate it:

```
var testKey = bytes.Repeat([]byte{1}, shift.BlockSize)
```

(Listing shift/7)

This, as I'm sure you already figured out, creates a key equivalent to (in hex):

We could have written this key out as a []byte literal, or as a hex string and then decoded it, but that all seems like a lot of typing. Instead, bytes.Repeat is a convenient way of constructing a long slice of non-zero bytes.

And, assuming we correctly apply this key, we should end up with a ciphertext where every byte value is one greater (modulo 256) than the plaintext:

```
var cipherCases = []struct {
    plaintext, ciphertext []byte
}{

{
    plaintext: []byte{0, 1, 2, 3, 4, 5},
    ciphertext: []byte{1, 2, 3, 4, 5, 6},
},
}
```

([Listing shift/7](#))

That'll do for now; we can add more cases later.

Creating the cipher object

Next, we're saying that in order to do any enciphering, we need to create a cipher object. Since the key is the same for all cases, so is the cipher. Accordingly, we can create it just once, before entering the case loop:

```
block, err := shift.NewCipher(key)
if err != nil {
    t.Fatal(err)
}
```

Now we can loop over each case, enciphering it with `block.Encrypt`:

```
got := make([]byte, len(tc.plaintext))
block.Encrypt(got, tc.plaintext)
```

And then we'll compare `want` and `got` in the usual way.

A test for Encrypt

So here's a test that should do the job:

```
func TestEncrypt(t *testing.T) {
    t.Parallel()
    block, err := shift.NewCipher(testKey)
    if err != nil {
```

```

        t.Fatal(err)
    }
    for _, tc := range cipherCases {
        name := fmt.Sprintf("%x + %x = %x", tc.plaintext,
testKey,
            tc.ciphertext)
        t.Run(name, func(t *testing.T) {
            got := make([]byte, len(tc.plaintext))
            block.Encrypt(got, tc.plaintext)
            if !bytes.Equal(tc.ciphertext, got) {
                t.Errorf("want %x, got %x", tc.ciphertext, got)
            }
        })
    }
}

```

([Listing shift/7](#))

The test for Decrypt will be very similar, as you'd expect:

```

func TestDecrypt(t *testing.T) {
    t.Parallel()
    block, err := shift.NewCipher(testKey)
    if err != nil {
        t.Fatal(err)
    }
    for _, tc := range cipherCases {
        name := fmt.Sprintf("%x - %x = %x", tc.ciphertext,
testKey,
            tc.plaintext)
        t.Run(name, func(t *testing.T) {
            got := make([]byte, len(tc.ciphertext))

```

```
        block.Decrypt(got, tc.ciphertext)
        if !bytes.Equal(tc.plaintext, got) {
            t.Errorf("want %x, got %x", tc.plaintext, got)
        }
    })
}
}
```

([Listing shift/7](#))

By the way, we won't need the tests for our old `Encipher` and `Decipher` functions any more, since we're replacing them with the new block-oriented code. And we can also delete or comment out the `crack` function and its test, since it won't work with the latest cipher code (we'll update this later).

So now we can turn to writing the `Encrypt` and `Decrypt` methods themselves, and (not surprisingly) that's pretty straightforward. Again, these will replace `Encipher` and `Decipher`, and we already did the substantive work on those (or rather, you did).

Writing `Encrypt` and `Decrypt`

In fact, we need *less* code than we did previously. Since the result slice is now passed in, we don't have to create it.

This is all that's now required:

```
func (c *shiftCipher) Encrypt(dst, src []byte) {
    for i, b := range src {
        dst[i] = b + c.key[i]
    }
}

func (c *shiftCipher) Decrypt(dst, src []byte) {
```

```
    for i, b := range src {
        dst[i] = b - c.key[i]
    }
}
```

([Listing shift/7](#))

Are we done? Almost! We missed one thing, but Go picked it up:

```
cannot use &shiftCipher{} (value of type *shiftCipher) as
cipher.Block value in return statement: *shiftCipher does not
implement cipher.Block (missing method BlockSize)
```

That looks a little scary, but you know what to do in this situation. Keep calm, grab a snack, and then read the message carefully from beginning to end. Let's break it down.

First of all, the message is about this `return` statement, in `NewCipher`:

```
return &shiftCipher{
    key: [BlockSize]byte(key),
}, nil
```

([Listing shift/7](#))

And the error is saying that the function promised to return a `cipher.Block`, which as we know is an interface, but what it *actually* returns is a pointer to a `shiftCipher` struct.

That's okay, as long as `*shiftCipher` *implements* `cipher.Block`, but Go is telling us that it doesn't:

`*shiftCipher` does not implement `cipher.Block`

We hoped it would; indeed, that was the plan, but apparently we're not quite there yet. Here's the problem:

```
missing method BlockSize
```

So, as often happens, the actual code problem is *not* with the `return` statement, which is perfectly correct. The problem is somewhere else: namely, that we haven't provided anywhere a `BlockSize` method on our struct type.

It's tempting, when you see a message like this, to drop into stochastic debugging mode and start randomly *frobbing* the `return` statement until it works. But we already know that's not a good strategy.

Instead, careful reading of the error message, tells us exactly what we need to do: implement the `BlockSize` method.

Forgotten something?

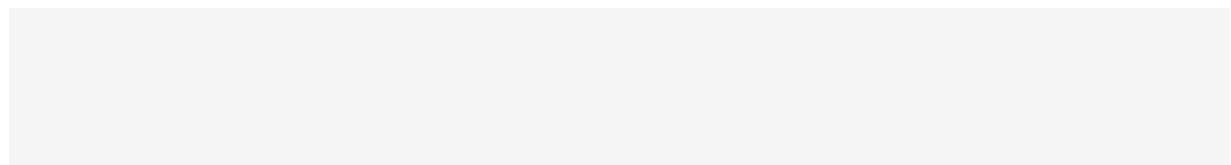
By the way, this is one nice side-effect of declaring `NewCipher`'s result type as `cipher.Block`, instead of `*shiftCipher`. Go can catch cases like this where we didn't finish implementing the interface properly. Over to you to fix it, then!

GOAL: Write `BlockSize`, guided by a test.

HINT: The `BlockSize` method is supposed to report the block size of the cipher object we call it on, and we know that this is always a fixed value: namely, `shift.BlockSize`. So that's what `BlockSize` should always return.

It's not hard to implement, and it's only slightly more difficult to test.

SOLUTION: All right, we know what the result of `BlockSize` should be, but first we need a cipher object to call it on. That means calling `NewCipher`. The rest is straightforward:



```
func TestBlockSize_ReturnsBlockSize(t *testing.T) {
    t.Parallel()
    block, err := shift.NewCipher(make([]byte, shift.BlockSize))
    if err != nil {
        t.Fatal(err)
    }
    want := shift.BlockSize
    got := block.BlockSize()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

([Listing shift/7](#))

The implementation is just a little bit of paperwork:

```
func (c shiftCipher) BlockSize() int {
    return BlockSize
}
```

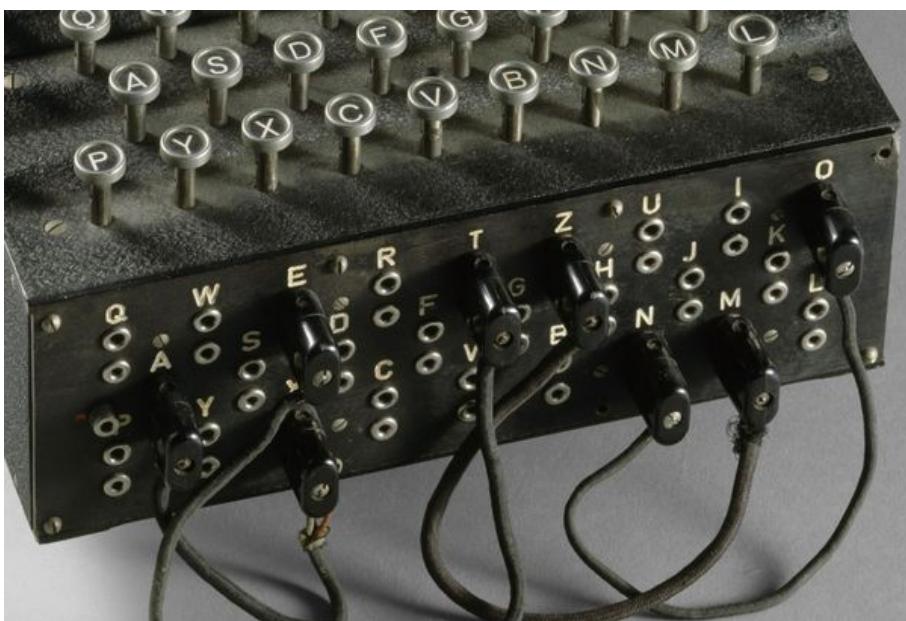
([Listing shift/7](#))

Nicely done! But don't stop now: we just broke the command-line tools that rely on `Encipher` and `Decipher`, so the next step is to get those working again.

9. Modes

The mantra of any good security engineer is: “Security is not a product, but a process.” It’s more than designing strong cryptography into a system; it’s designing the entire system such that all security measures, including cryptography, work together.

—Bruce Schneier, [“Risks of Relying on Cryptography”](#)



(photo by [Pascal Segrette](#))

We've made some great progress so far. We've taken our shift cipher and upgraded it to support the `cipher.Block` interface. This will enable us to do some interesting things with it later on.

Handling data in blocks

But first, let's revisit our command-line tools to bring them up to date with the new API. These are the programs that Alice and Bob will actually run to encipher and decipher messages, and we'll now need to make a few modifications to call the new cipher code in the right way.

Updating the encipher tool

Here's the code we wrote previously for the encipher tool, for example:

```
func main() {
    keyHex := flag.String("key", "01",
        "key in hexadecimal (for example 'FF')")
    flag.Parse()
    plaintext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    ciphertext := shift.Encipher(plaintext, key)
    os.Stdout.Write(ciphertext)
}
```

([Listing shift/6](#))

Fine, but there's no longer an `Encipher` function we can call directly: we need to create a `block` object first:

```
block, err := shift.NewCipher(key)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
```

```
    os.Exit(1)
}
```

Because `NewCipher` already checks that the key is the right size, we needn't worry about doing that again here. Instead, we can just report any error to Alice and let her deal with it.

Block by block

Next, we'll need to call the `Encrypt` method on our block cipher, to encipher Alice's plaintext. But there's one problem: because it's now a *block* cipher, we can only pass a block's worth of data to `Encrypt` at a time.

We want to deal with the data in chunks of 32 bytes, in other words. And maybe it happens that `plaintext` *is* exactly that long. That would be convenient, but unlikely. Most of the time the message to be enciphered will be either longer or shorter than our block size.

We need some code that can take data of any length, and encipher it a block at a time, using our `block` object. In principle, this is quite straightforward: we just keep reading 32-byte chunks of the data and enciphering them until we've read all the data.

The `cipher.BlockMode` interface

Again, the `cipher` package provides a useful interface for us to implement here:

```
type BlockMode interface {
    BlockSize() int
    CryptBlocks(dst, src []byte)
}
```

([crypto/cipher](#))

The `BlockSize` method is the same as for `cipher.Block`, but we now also have `cryptBlocks`, which takes a `src` slice of arbitrary length, and enciphers it, blockwise, into `dst`, in the way that we described.

The specific way in which we choose to encipher successive blocks of data is called the *mode of operation* for the cipher. There are several modes used in practice, but first let's consider the simplest mode imaginable: just encipher each block independently.

A simple block mode

To do this, we need something that implements the `BlockMode` interface, so again it's over to you:

GOAL: Write a test for a new type that implements `cipher.BlockMode`, and that enciphers data using the shift cipher.

HINT: Let's walk through what we need for a `BlockMode` implementation, step by step.

Just as with `cipher.Block`, we'll define some struct type that will implement the interface. Note that, with a `BlockMode`, we no longer have separate `Encrypt` and `Decrypt` methods. Instead, they're replaced by a single method: `CryptBlocks`.

This implies that we'll need *two* types, both implementing `BlockMode`: one for encrypting, and another for decrypting. They'll be almost exactly the same, as you'd expect: it's just that their `cryptBlocks` methods will work in opposite "directions".

Let's start with the encrypter. Indeed, `encrypter` seems the most obvious name to give this type. I'm a big fan of obviousness-oriented programming, aren't you?

We'll need a type definition for this struct, and a constructor for it (`NewEncrypter`). And we know, from the interface definition, that we'll need to add a `CryptBlocks` method to our new type.

We also know that the encrypter will need to *use* a given block cipher to do the actual enciphering. So a decent test for all this should do something like the following:

1. Create a new cipher object
2. Use it to create a new encrypter
3. Use the encrypter's `CryptBlocks` method to encipher some data
4. Check the result

Testing `CryptBlocks`

SOLUTION: Here's a test that fulfils these requirements:

```
func TestEncrypterEnciphersBlockAlignedMessage(t *testing.T) {
    t.Parallel()
    plaintext := []byte("This message is exactly 32 bytes")
    block, err := shift.NewCipher(testKey)
    if err != nil {
        t.Fatal(err)
    }
    enc := shift.NewEncrypter(block)
    want := []byte("Uijt!nfttbhf!jt!fybdumz!43!czuft")
    got := make([]byte, 32)
    enc.CryptBlocks(got, plaintext)
    if !bytes.Equal(want, got) {
        t.Errorf("want %v, got %v", want, got)
    }
}
```

([Listing shift/7](#))

Straightforward, right? We encipher a 32-byte message and check the result.

With the test to guide us, we're ready to write `encrypter`. Over to you for this part.

GOAL: Implement `encrypter`.

HINT: Just as with the `shiftCipher` object, we'll need some struct type, with the right methods, and a suitable constructor. That's fairly straightforward, but implementing the `cryptBlocks` method might need a little more thought.

We don't actually need to write any code to do the enciphering. That's already taken care of by the `shiftCipher` object. We can pass our plaintext to the cipher's `Encrypt` method, one block at a time.

And, since there's probably more than one block's worth of plaintext, that sounds like a job for a loop, doesn't it? We'll want to loop over the plaintext a block at a time, enciphering each block and writing the result to our destination slice, until there are no more blocks.

As a warm-up for this problem, you might like to try the following slightly simpler exercise. Write a Go function that takes a slice of bytes and prints it in groups of five bytes at a time, one group per line. If you can do this, you can write `cryptBlocks`! (Or take a peek at listing [groups/1](#).)

SOLUTION: Let's start with the struct type definition. What fields do we need?

Well, one of the nice things about `cipher.BlockMode` is that it's entirely independent of the specific `cipher` in use. A cipher mode says, in effect, "You give me some block cipher, and I'll use it to encipher your data for you, blockwise."

That suggests that we could store the `cipher.Block` itself on the struct. While we're at it, let's add a field for the block size, too:

```
type encrypter struct {
    block      cipher.Block
    blockSize int
}
```

([Listing shift/7](#))

Sorry about the confusing name: `block` here means, as before, not some chunk of data, but some block *cipher*. Calling the cipher object `block` is weird, but the standard library code uses this name everywhere, so let's follow its example.

Creating the encrypter object

Just as before, since we've defined some new type, we'll need a constructor for it:

```
func NewEncrypter(block cipher.Block) cipher.BlockMode {
    return &encrypter{
        block:      block,
        blockSize: block.BlockSize(),
    }
}
```

([Listing shift/7](#))

Note that we *don't* hard-wire `blocksize` to be `shift.BlockSize`: that would be incorrect. Since we're saying that, in principle, an `encrypter` can work with any block cipher, we can't know its block size in advance.

Instead, we need to ask the `block` object, "Hey, what's your block size, by the way?" To do that, we call `block.BlockSize`.

Implementing CryptBlocks

Now let's consider how to write `CryptBlocks`. Here's the signature we need:

```
func (e *encrypter) CryptBlocks(dst, src []byte)
```

As we know, the problem of enciphering any given block is already solved: we can just pass it to `e.block.Encrypt`. So all we need to do is figure out how to take the input one block at a time.

Given some byte slice, such as `src`, it's straightforward to write an index expression that extracts the first `n` bytes:

```
src[:n]
```

And in fact, we know what `n` we want in this case: it's the block size. So we can write:

```
e.block.Encrypt(dst[:e.blockSize], src[:e.blockSize])
```

Having enciphered the first `blockSize` bytes of `src` into `dst`, we should then *re-slice* both `src` and `dst` to remove the bytes we've already processed:

```
src = src[e.blockSize:]
dst = dst[e.blockSize:]
```

So far, so good. We've enciphered the first block. But, assuming there are more blocks to come, we'll need some kind of loop. When should the loop

terminate? That's easy: when there's no data left in `src`.

Here's the complete `CryptBlocks` method so far, then:

```
func (e *encrypter) CryptBlocks(dst, src []byte) {
    for len(src) > 0 {
        e.block.Encrypt(dst[:e.blockSize], src[:e.blockSize])
        src = src[e.blockSize:]
        dst = dst[e.blockSize:]
    }
}
```

We can read this as saying “Keep chopping block-sized pieces off the plaintext and enciphering them until there are no more pieces.”

You might wonder what happens if the length of `src` isn't an exact multiple of the block size. Won't we end up trying to encipher an incomplete block in that case?

That's true, but to simplify things, let's say that `CryptBlocks` only has to deal with data that's *block-aligned*—that is, data that can be split into an exact number of blocks, with no leftovers. We'll see how to arrange this later.

Phew! That's the hard part over with. But there's one more thing we need to complete the implementation of `BlockMode`. Luckily, it's an easy one:

```
func TestEncrypterCorrectlyReportsCipherBlockSize(t *testing.T) {
    t.Parallel()
    block, err := shift.NewCipher(testKey)
    if err != nil {
        t.Fatal(err)
    }
}
```

```
    }
    enc := shift.NewEncrypter(block)
    want := shift.BlockSize
    got := enc.BlockSize()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}
```

([Listing shift/7](#))

And there's no substantive code for us to write, since we already made the sensible decision to store the block size in the `encrypter` struct itself. This method is simply an accessor for that field:

```
func (e encrypter) BlockSize() int {
    return e.blockSize
}
```

([Listing shift/7](#))

Great! This passes the test, so let's see if we can plug our new code into the `encipher` tool.

Using a `BlockMode` in `encipher`

We've already decoded Alice's key and used it to create a cipher, so we can now take *that* and use it to create a `BlockMode`. Then enciphering is as simple as passing the plaintext to `CryptBlocks`.

Here's the complete `main` function for `encipher`:

```

func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := shift.NewCipher(key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    enc := shift.NewEncrypter(block)
    plaintext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    ciphertext := make([]byte, len(plaintext))
    enc.CryptBlocks(ciphertext, plaintext)
    os.Stdout.Write(ciphertext)
}

```

It's long, but it's not complicated: it's just plugging together everything that we've developed so far. We get the key from Alice, use it to construct a block cipher, then use *that* to construct an encrypter.

Let's try it out:

```
echo "This is 32 bytes, including EOF" | go run ./cmd/encipher -key \
```

Uijt!jt!43!czuft-!jodmvejoh!FPG

Super nice. But you might have noticed that both our test messages were exactly 32 bytes, which isn't a coincidence: that's the same as our block size.

So what would happen if we tried to encipher something that's *not* an exact multiple of 32 bytes in length? Let's find out:

```
echo "This is 31 long, including EOF" | go run ./cmd/encipher -key \
```

```
panic: runtime error: slice bounds out of range [:32] with capacity 31
```

```
goroutine 1 [running]:  
github.com/bitfield/shift.(*encrypter).CryptBlocks(...)  
    shift.go:67 +0x130
```

Dang it!

Block alignment

We wrote `cryptBlocks` under the explicit assumption that it'll have to deal only with block-aligned data, remember, so it's not really a surprise that something blew up here.

But what *precisely* is the issue that we need to fix? Here's the `cryptBlocks` loop again:

```
for len(src) > 0 {  
    e.block.Encrypt(dst[:e.blockSize], src[:e.blockSize])  
    ...  
}
```

When misaligned data attacks

We're trying to slice the first 32 bytes off `src` (and `dst`) but in our example case, there *are* only 31 bytes, so this index expression panics:

```
src[:e.blockSize]
```

What can we do? Any time the data isn't block-aligned, like now, we'll end up with a "short block" at the end. And that's going to be true the majority of the time, so we can't ignore the problem.

Checking our assumptions

We could probably figure out how to *check* whether `src` is block-aligned, but what if it turns out that it isn't? What do we do then?

We can't return an error, because the signature of `cryptBlocks` is fixed by the `cipher.BlockMode` interface, and it doesn't include an error result.

But that's okay. The lack of any provision for an error *implies* that `cryptBlocks` should only be used on block-aligned data. If it's called with unsuitable data, then that means the programmer has done something wrong (in this case, that's us).

The conventional way to signal that kind of problem in Go is to call `panic`, so let's do that:

```
if len(src)%e.blockSize != 0 {  
    panic("encrypter: input not full blocks")  
}
```

([Listing shift/7](#))

You might object that we don't need this extra check, because the code *already* panics in this case! That's true, but the message, while strictly accurate, doesn't help the programmer figure out what's wrong:

```
panic: runtime error: slice bounds out of range [:32] with capacity 31
```

Instead, by calling their attention to the underlying problem ("input not full blocks"), this panic can be helpful.

While we're thinking about possible panic situations, there's another one. Can you spot it? What other implicit assumptions is our code making about its input?

The long and short of it

This isn't so easy to see, but we're assuming that `dst` is at least as long as `src`. The block cipher's `Encrypt` method loops over `src`, enciphering each byte and writing it to the corresponding index of `dst`. So if `dst` is too short, eventually that write will panic:

```
panic: runtime error: index out of range [X] with length Y
```

Again, not super helpful. Let's add value by checking the slice lengths and pre-emptively panicking with a more informative message:

```
if len(dst) < len(src) {
    panic("encrypter: output smaller than input")
}
```

Should we add tests for these two cases? Well, that's not quite straightforward. First, could we even *catch* a panic in a test?

A test about nothing

Yes, we could, by writing something like this:

```
defer func() {
    if r := recover(); r != nil {
        t.Fatal(r)
    }
}()
```

This isn't necessary, of course, since the test *already* fails if it panics: we don't need to write special code for that. The bigger problem, though, is that we're saying this code is *supposed* to panic! In fact, we'd want to fail the test if it *didn't*:

```
defer func() {
    if r := recover(); r == nil {
        t.Fatal("expected panic, but came there none")
    }
}()
```

Fine. We can write a test for “`CryptBlocks` panics when input data is not block-aligned”, using this `recover` code, and it will pass. The problem is that is if we now comment out the check that `len(dst) < len(src)`, it will *still* pass! So what can it really be testing?

We already know the code panics with a “slice bounds out of range” message when we don’t catch the misalignment. Thus, there’ll be a panic either way, meaning that the test can’t tell us anything about whether or not `CryptBlocks` sanity-checks its input.

So it doesn’t usually make much sense to try to test for panics (though it’s nice that we know how to do it, if we really want to). Panics that are likely to affect users will pretty soon be noticed when we run the program anyway.

A panic-proof `CryptBlocks`

Here’s the updated version of `CryptBlocks`, then, now with extra sanity checks on its input:

```
func (e *encrypter) CryptBlocks(dst, src []byte) {
    if len(src)%e.blockSize != 0 {
        panic("encrypter: input not full blocks")
    }
    if len(dst) < len(src) {
        panic("encrypter: output smaller than input")
    }
    for len(src) > 0 {
        e.block.Encrypt(dst[:e.blockSize], src[:e.blockSize])
        src = src[e.blockSize:]
        dst = dst[e.blockSize:]
    }
}
```

(Listing shift/7)

Let's check that our new, more helpful, error message is shown when we try to encipher dodgy data:

```
echo "This is 31 long, including EOF" | go run ./cmd/encipher -key \
```

panic: encrypter: input not full blocks

Of course, we'd prefer not to have to show an error at all. Only one in every 32 messages will be block-aligned by chance, statistically speaking, and that's a pretty low success rate for an encryption tool.

To get around this, we'll need some way of ensuring that even if Alice's message doesn't end on a block boundary, the plaintext we actually encipher always will.

10. Padding

“Where is, repeat, where is Task Force Thirty Four? The world wonders.”

—[The World Wonders](#)



We've nearly got enciphering using our super-whizzy block cipher working nicely. The only remaining problem is what to do when the input message is not block-aligned.

There's a fairly straightforward solution. If the plaintext doesn't already end on a block boundary, we can add some *padding* bytes. These will fill out the data to a complete block, before it gets passed to cryptBlocks.

A padding scheme

So, what bytes should we use as padding data? Our first idea might be to just add the requisite number of *zero* bytes, since their actual values don't matter.

The problem with this idea is, what happens at the other end of the process?

Making ends meet

In other words, suppose we’re decrypting a ciphertext, and we find that the last few bytes of the plaintext are zeroes, something like this:

```
DE AD BE EF 00 00 00 00 00
```

The question is, are these trailing zero bytes part of the original plaintext, or were they added as padding? There’s no way to know.

We could just declare that the plaintext is not *allowed* to end with a sequence of zero bytes, but that sounds awful. There might be lots of such plaintexts, and we simply wouldn’t be able to encipher them using this system.

The “illegal pixel” problem

We wouldn’t want to have to tell Alice “Sorry, the bottom right-hand pixel in your image is black, and that’s not allowed!” Users can get a bit tetchy about things like this, so we’ll need to think of a better padding scheme.

GOAL: Try to work out a padding scheme where the padding bytes can always be unambiguously removed after deciphering.

HINT: What if messages were *always* padded, no matter whether they end on a block boundary or not?

SOLUTION: If we *always* pad messages, even when we don’t strictly need to, then the “illegal pixel” problem doesn’t arise. We don’t have to decide whether or not there are any padding bytes to remove, because the answer is always yes. All we need to know is how many.

An unambiguous padding scheme

Here's a fairly simple scheme that would work. Given some data that's N bytes short of a full block, we add N padding bytes, and we set each of those bytes to the value N .

In other words, supposing the data is three bytes short of a block, we pad it with three bytes, all with the value 03:

```
03 03 03
```

If the message happens to end exactly on a block boundary (which it will about 3% of the time), then we'll add a full block's worth of padding (32 copies of the byte value 32, in our case).

We have to transmit the padding along with the message, of course, and so we're wasting a little space, but never more than a block's worth per message. That's a small price to pay for guaranteed block alignment.

Now *unpadding* is easy. We know there will always be padding, and the last byte of the data will tell us how much. If the data ends with a 3 byte, then strip the last three bytes. If it ends with a 32 byte, then we strip 32 bytes.

Testing Pad

We're now ready to write a `Pad` function that ensures our input plaintext is block-aligned, and after that we'll tackle a corresponding `Unpad` function to remove any extraneous padding from decrypted data. Over to you!

GOAL: Write a test for `Pad`.

HINT: As usual, we'll need to write some new test cases. We could, of course, cover every possible length of input, from zero to `BlockSize`. But that would be *slightly* overdoing it.

In order to have reasonable confidence that our implementation of `Pad` and `Unpad` works, we only need a sample of plausible inputs, as long as we include the *interesting* ones.

Some interesting inputs to try might be, for example:

- Data that is just short of a full block, by a few bytes
 - Data that ends exactly on a block boundary
 - No data at all
-

SOLUTION: First, let's design our test case struct:

```
var padCases = []struct {
    name      string
    raw, padded []byte
}{
```

It would be tedious to write out literals of 32-byte blocks, so let's use a much smaller block size for the tests: 4 bytes, let's say. Our padding scheme doesn't depend on the exact block size in use, so we'll make that an input to the `Pad` function.

Again, we didn't name the fields `want` and `got`, because we'll want to use the same cases to test both padding and unpadding. So we've chosen `raw` for the unpadded text and `padded` for the padded equivalent.

If the data is one byte short of a full block, we know that `Pad` should add a single trailing `1` byte (and `Unpad` should remove it). So here's that case:

```
{
    name: "1 short of full block",
    raw: []byte{0, 0, 0},
    padded: []byte{0, 0, 0, 1},
},
```

Similarly, if the data is two bytes short, we should add two 2 bytes, and if it's three short, we should add three 3 bytes:

```
{
    name: "2 short of full block",
    raw: []byte{0, 0},
    padded: []byte{0, 0, 2, 2},
},
{
    name: "3 short of full block",
    raw: []byte{0},
    padded: []byte{0, 3, 3, 3},
},
```

So far, so straightforward. Now comes the interesting case: data that's *already* block-aligned. We're saying that `Pad` should add a full block of four 4 bytes:

```
{
    name: "full block",
    raw: []byte{0, 0, 0, 0},
    padded: []byte{0, 0, 0, 0, 4, 4, 4, 4},
},
```

And, just for completeness, let's consider the case where there's no data at all. This, too, is block-aligned, and again we should add a full block of padding:

```
{  
    name: "empty block",  
    raw: []byte{},  
    padded: []byte{4, 4, 4, 4},  
,
```

Here's a test that checks these cases:

```
func TestPad(t *testing.T) {  
    t.Parallel()  
    blockSize := 4  
    for _, tc := range padCases {  
        t.Run(tc.name, func(t *testing.T) {  
            got := shift.Pad(tc.raw, blockSize)  
            if !bytes.Equal(tc.padded, got) {  
                t.Errorf("want %v, got %v", tc.padded, got)  
            }  
        })  
    }  
}
```

([Listing shift/7](#))

Implementing Pad

GOAL: Implement Pad so that your test passes.

HINT: The signature of the Pad function is implied by the test:

```
func Pad(data []byte, blockSize int) []byte
```

So what does it need to do ? We can break the work down into three sub-tasks:

1. Determine how many padding bytes we need
 2. Generate a slice of that many bytes
 3. Append it to the data and return the result
-

SOLUTION: Let's take each of these tasks in order.

First, how many padding bytes do we need? That depends on the length of the last (possibly incomplete) block of data, and *that* sounds like a job for our friend the remainder operator:

```
len(data)%blockSize
```

In other words, if we take the remainder of `len(data)` after dividing it by `blockSize`, we get the number of “trailing” bytes. To know how many we need to *add* to make up a full block, we can subtract this number from `blockSize`:

```
n := blockSize - len(data)%blockSize
```

Done! Next job: generate the padding data. That's easy now that we know `n` (the number of bytes by which `data` is short).

Suppose `n` is 3, for example, then we need to generate a slice of three bytes where every element is 3. The `bytes.Repeat` function is handy for this:

```
padding := bytes.Repeat([]byte{byte(n)}, n)
```

Finally, we append this to `data` and return the result. So here's the complete function:

```
func Pad(data []byte, blockSize int) []byte {
    n := blockSize - len(data)%blockSize
    padding := bytes.Repeat([]byte{byte(n)}, n)
    return append(data, padding...)
}
```

([Listing shift/7](#))

Writing Unpad

GOAL: Implement `Unpad`, guided by a test.

HINT: Padding was a *little* fiddly, to be sure, but unpadding is much easier. We know that the last byte of the data tells us `n`: how many trailing bytes of padding need to be stripped. Once we've done that, we can return the remaining data.

SOLUTION: The test is, of course, the exact inverse of the test for `Pad`:

```
func TestUnpad(t *testing.T) {
    t.Parallel()
    blockSize := 4
    for _, tc := range padCases {
        t.Run(tc.name, func(t *testing.T) {
            got := shift.Unpad(tc.padded, blockSize)
            if !bytes.Equal(tc.raw, got) {
                t.Errorf("want %v, got %v", tc.raw, got)
            }
        })
    }
}
```

```
        }
    })
}
}
```

([Listing shift/7](#))

Now let's tackle the implementation. First, we should look at the last byte of the data, to find out what `n` is:

```
n := int(data[len(data)-1])
```

Once we know `n`, we can use an index expression to return all but the last `n` bytes of `data`:

```
return data[:len(data)-n]
```

So here's a working version of `Unpad`, in all its terse glory:

```
func Unpad(data []byte, blockSize int) []byte {
    n := int(data[len(data)-1])
    return data[:len(data)-n]
}
```

([Listing shift/7](#))

This padding scheme is pleasantly simple to describe and implement (it's known as "PKCS #7 padding"), and it's documented in [RFC 2315](#), which also notes:

This padding method is well-defined if and only if [block size] $k < 256$ [bytes]; methods for larger k are an open issue for further study.

That makes sense: we can't use block sizes bigger than we can represent in a single byte, which means this scheme wouldn't work as-is with blocks of more than 255 bytes.

Such large block sizes are rare in practice, though, and if we ever need to handle them, we can always use another padding scheme. This one will do for now.

Padding input to encipher

Now that we have a working Pad and Unpad, we can modify our encipher tool to pad the data on the way in, by adding this line:

```
plaintext = shift.Pad(plaintext, enc.BlockSize())
```

Adding the padding

So here's the complete code for encipher, including padding:

```
func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := shift.NewCipher(key)
```

```
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

enc := shift.NewEncrypter(block)
plaintext, err := io.ReadAll(os.Stdin)

if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

plaintext = shift.Pad(plaintext, enc.BlockSize())
ciphertext := make([]byte, len(plaintext))
enc.CryptBlocks(ciphertext, plaintext)
os.Stdout.Write(ciphertext)

}
```

(Listing shift/7)

Checking the results

Let's try it out with our previous just-short-of-a-block message:

Uijt!it!42!mpoh-!jodmvejoh!FPG

Very neat. Our “tiger” text is 468 bytes long, which is not an exact multiple of the blocksize, so we should now be able to handle this too:

If the padding has worked correctly, then we should end up with a ciphertext file that *is* an exact multiple of 32 bytes long:

```
wc -c enciphered.bin
```

480 enciphered.bin

The original file was 468 bytes, and we now have 480, so the padding process has added 12 extra bytes. If this has been done correctly, then each of those padding bytes should have the value `0D` (12 in hexadecimal). Let's see:

```
hexdump enciphered.bin | tail -n 2
```

000001d0 e3 81 9e 0b 0d 0d

Nailed it!

Deciphering

The next question is, can we *decipher* this file again?

We'll need to adapt the existing decipher tool to take account of padding, and it also makes sense that we'd use some implementation of

`cipher.BlockMode` for the actual deciphering.

Adding the unpadding

Well, here's the code we'd *like* to write for `decipher`, including the necessary call to `Unpad` before printing the plaintext:

```
func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := shift.NewCipher(key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    ciphertext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    plaintext := make([]byte, len(ciphertext))
    dec := shift.NewDecrypter(block)
    dec.CryptBlocks(plaintext, ciphertext)
    plaintext = shift.Unpad(plaintext, shift.BlockSize)
    os.Stdout.Write(plaintext)
}
```

([Listing shift/7](#))

As you can see, it looks a lot like the `encipher` tool, except that instead of calling `NewEncrypter`, naturally enough, we call `NewDecrypter`.

There's just one problem: we don't actually have that function yet.

A decrypter that implements `BlockMode`

We're saying, in effect, we wish we had a `NewDecrypter` function that would return a `decrypter` object, because if we had one of those we could use it to `cryptBlocks` the padded ciphertext and retrieve the plaintext.

GOAL: Write the necessary code to get this program working.

HINT: Just as in previous versions of the code, we can treat deciphering as the exact inverse of enciphering. Therefore, if we had some type `decrypter` that implemented the `cipher.BlockMode` interface, it would look very similar to the existing `encrypter`.

The only difference, indeed, is that it would have to call the `Decrypt` method on its block cipher, instead of `Encrypt`. Can you see what to do?

SOLUTION: We needn't do anything fancy here. We can just copy and paste the `encrypter` code and tests, replacing `Encrypt` with `Decrypt` at the appropriate point. Duplication is better than complication, as we've already seen in other contexts.

We'll start with the tests:

```
func TestDecrypterDeciphersBlockAlignedMessage(t *testing.T) {
    t.Parallel()
    ciphertext := []byte("Uijt!nfttbhf!jt!fybdumz!43!czuft")
    block, err := shift.NewCipher(testKey)
```

```

    if err != nil {
        t.Fatal(err)
    }
    dec := shift.NewDecrypter(block)
    want := []byte("This message is exactly 32 bytes")
    got := make([]byte, 32)
    dec.CryptBlocks(got, ciphertext)
    if !bytes.Equal(want, got) {
        t.Errorf("want %v, got %v", want, got)
    }
}

func TestDecrypterCorrectlyReportsCipherBlockSize(t *testing.T) {
    t.Parallel()
    block, err := shift.NewCipher(testKey)
    if err != nil {
        t.Fatal(err)
    }
    dec := shift.NewDecrypter(block)
    want := shift.BlockSize
    got := dec.BlockSize()
    if want != got {
        t.Errorf("want %d, got %d", want, got)
    }
}

```

([Listing shift/7](#))

Now the implementation (which, as predicted, is nearly identical to encrypter):

```

type decrypter struct {
    block      cipher.Block
    blockSize int
}

func NewDecrypter(block cipher.Block) cipher.BlockMode {
    return &decrypter{
        block:      block,
        blockSize: block.BlockSize(),
    }
}

func (e decrypter) BlockSize() int {
    return e.blockSize
}

func (e *decrypter) CryptBlocks(dst, src []byte) {
    if len(src)%e.blockSize != 0 {
        panic("decrypter: input not full blocks")
    }
    if len(dst) < len(src) {
        panic("decrypter: output smaller than input")
    }
    for len(src) > 0 {
        e.block.Decrypt(dst[:e.blockSize], src[:e.blockSize])
        src = src[e.blockSize:]
        dst = dst[e.blockSize:]
    }
}

```

([Listing shift/7](#))

This makes sense, doesn't it? The decrypter is literally almost line-for-line identical to the encrypter, except that it calls the block cipher's Decrypt

method instead of `Encrypt`. The rest is just the boilerplate required to be a `cipher.BlockMode`.

Waiting for the tiger

If we've got it right, then, we should be able to decipher the message we enciphered earlier, and recover the original tigertext:

The tiger appears at its own pleasure. When we become very silent
...
...

Great! It's taken us a while to get back to this point, and in the meantime we've updated our cipher to be a proper, grown-up block cipher with strong keys. We've also implemented a very simple operating mode (a bit too simple, as we'll see later), and put in the necessary padding and unpadding code to block-align our data.

11. Enumeration

Brute-force attacks against 256-bit keys will be infeasible until computers are built from something other than matter and occupy something other than space.

—Bruce Schneier, “[Applied Cryptography](#)”



(photo by [CBtronica](#))

Let’s return now to the question of cracking. The previous version of crack that we wrote worked by calling the `Decipher` function directly, but we don’t have that anymore. Instead, we’ll need to call `NewCipher` to construct a block cipher with the key we’re trying, then call its `Decrypt` block to check the result.

Adventures in keyspace

We’ll also need to enumerate a lot more possible keys, since the current version of the shift cipher uses a fixed-size 32-byte key. The principle is the same as we used before when brute-forcing, but we’ll just need *more* brute force.

A new test for cracking long keys

Let's start by updating the test. Here's what we have right now:

```
func TestCrack(t *testing.T) {
    t.Parallel()
    for _, tc := range cases {
        name := fmt.Sprintf("%s + %d = %s", tc.plaintext, tc.key,
            tc.ciphertext)
        t.Run(name, func(t *testing.T) {
            got, err := shift.Crack(tc.ciphertext,
                tc.plaintext[:3])
            if err != nil {
                t.Fatal(err)
            }
            if tc.key != got {
                t.Fatalf("want %d, got %d", tc.key, got)
            }
        })
    }
}
```

We can't use our old test cases, since they don't have 32-byte keys. Let's just write a new test for something enciphered using our full-length `testKey`:

```
func TestCrack(t *testing.T) {
    t.Parallel()
    plaintext := []byte("This message is exactly 32 bytes")
    ciphertext := []byte("Uijt!nfttbhf!jt!fybdumz!43!czuft")
    want := testKey
    got, err := shift.Crack(ciphertext, plaintext)
    if err != nil {
```

```
    t.Fatal(err)
}
if !bytes.Equal(want, got) {
    t.Fatalf("want %d, got %d", want, got)
}
}
```

As with the tests for encrypter and decrypter, we needn't bother considering non-block-aligned message data. We already know we can handle that with padding. What matters is that crack can recover the test key.

A block is a black box

Let's look at our previous version:

```
func Crack(ciphertext, crib []byte) (key []byte, err error) {
    for k := range min(MaxKeyLen, len(ciphertext)) {
        for guess := range 256 {
            result := ciphertext[keyPos] - byte(guess)
            if result == crib[keyPos] {
                key = append(key, byte(guess))
                break
            }
        }
        if bytes.Equal(crib, Decipher(ciphertext[:len(crib)],
key)) {
            return key, nil
        }
    }
    return nil, errors.New("no key found")
}
```

This is sort of okay, but it makes some (now) unwarranted assumptions about the cipher scheme in use. Specifically, it assumes that given the N th byte of the ciphertext and the N th byte of the key, we can recover the N th byte of the plaintext.

The problem is that with a block cipher, we're only allowed to think about enciphering or deciphering a whole block at once. That's what it means to be a block cipher.

In effect, our cracking code is trying to "reach inside" the cipher's implementation and bypass the standard `cipher.Block` interface. But what if the cipher scheme involves shuffling parts of a block around before, after, or during enciphering?

In fact, many real cipher schemes do just this. So it's not, in principle, meaningful to ask what effect changing the key has on individual bytes in isolation. It happens to work with the shift cipher, but only by coincidence, because the shift cipher doesn't do any intra-block shuffling.

If we're to crack more realistic cipher schemes, we must take the API defined by `cipher.Block` seriously, and that means we're only allowed to test our guessed key against a whole block at a time.

GOAL: Update `Crack` to test its guesses by calling `block.Decrypt`, so that it no longer relies on specific knowledge about the cipher scheme in use.

HINT: This is really a two-part problem. Part 1 is generating all possible 32-byte keys. Part 2 is trying each key against a block of the ciphertext, to see if the result matches the crib.

Enumerating long keys

Key enumeration is simple in principle, but a little fiddly in practice. If the key were a Go `uint64`, for example, we could write something like this:

```
for key := uint64(0); key <= math.MaxUint64; key++ {  
    // translate 'key' value into a byte slice  
    // try key against ciphertext  
}
```

Unfortunately, a `uint64` is only 64 bits (8 bytes) wide, as the name suggests. Our keys are 256 bits (32 bytes), which is much too large to fit into any built-in Go numeric type.

There is a standard library `math/big` package for arbitrary-precision arithmetic, and we could use this, but let's first see if we can solve the problem by just twiddling bytes directly.

Suppose someone gave you a slice of 32 bytes representing a 256-bit number, and asked you what the next higher number would be. If you could answer that question, you could in principle iterate through all possible values representable by the slice. Does that give you any ideas?

Key *trying* is comparatively straightforward, since we already wrote code to decrypt ciphertexts using a block cipher object. The body of your key-trying loop would have to first create the cipher object using `key`, then call its `Decrypt` method with the first block of the ciphertext (because that's the only block we have a crib to check against).

Over to you!

SOLUTION: As we've just seen, one way to approach the key enumeration problem is to write some generalised `Next` function that takes the current key and tells you what value to try *next*. All the `Crack` function has to do, then, is start with the zero key, then keep calling `Next` and trying the result until it runs out of possibilities.

Designing a `Next` function

That last part also implies that `Next` can fail, when it can't increment the key any further without overflowing the number of bytes available. So let's

suppose `Next` has a signature something like:

```
func Next(key []byte) ([]byte, error) {
```

We don't need to hard-wire it to operate only on 32-byte arrays; in fact, taking a slice of any length means we can write a more concise test. Here's one that might work:

```
func TestNextCorrectlyIncrementsInputWithoutOverflow(t
*testing.T) {
    t.Parallel()
    tcs := []struct {
        input, want []byte
    }{
        {
            input: []byte{0, 0, 0},
            want:  []byte{1, 0, 0},
        },
        {
            input: []byte{255, 0, 0},
            want:  []byte{0, 1, 0},
        },
        {
            input: []byte{255, 255, 0},
            want:  []byte{0, 0, 1},
        },
        {
            input: []byte{255, 255, 254},
            want:  []byte{0, 0, 255},
        },
    }
}
```

```

    for _, tc := range tcs {
        t.Run(fmt.Sprintf("%x", tc.input), func(t *testing.T) {
            got, err := shift.Next(tc.input)
            if err != nil {
                t.Fatal(err)
            }
            if !bytes.Equal(tc.want, got) {
                t.Errorf("want %v, got %v", tc.want, got)
            }
        })
    }
}

```

Testing Next

Let's look at one of these test cases in detail:

```

input: []byte{0, 0, 0},
want:  []byte{1, 0, 0},

```

We're saying that calling `Next` on `input` should change the first byte from a zero to a one. Wait, what?

You might have expected that it would change the *last* byte, since we're used to seeing numbers where the digits are given in decreasing order of size. For example, in the decimal number “123”, the first digit is the most significant (representing 100), and the last is the least significant (representing 3).

Big endians and little endians

This way of ordering digits is known as *big-endian*, because we start with the big end of the number first. By contrast, then, *little-endian* byte order

would put the least significant byte first. For some applications, big-endianness makes sense, and for others, little-endianness is more appropriate.

Since we'll be looping over these bytes, it's just simpler in Go to start at the beginning of the slice and work forwards. That's why we construct the test cases this way. In fact, endianness is irrelevant for what we're doing, because we never said that we have to try all the keys *in a specific order*. So long as we end up generating every possible 32-byte slice, it's all good.

We've also said that `Next` needs to return an error when it *can't* generate the next key, because there are no more possible keys, so let's test that, too:

```
func TestNextReturnsErrorWhenNextKeyWouldOverflow(t *testing.T) {
    t.Parallel()
    _, err := shift.Next([]byte{255, 255, 255})
    if err == nil {
        t.Fatal("want error on key overflow, but got nil")
    }
}
```

A simple implementation

Here's one version that could work:

```
func Next(key []byte) ([]byte, error) {
    for i := range key {
        if key[i] < 255 {
            key[i]++
            return key, nil
        }
        key[i] = 0
    }
}
```

```
    }
    return nil, errors.New("overflow")
}
```

There are many other ways we could have done this, of course, and this one is by no means the most efficient way, but it does have the advantage of being fairly easy to write and understand. Here's how it works.

Given a particular key value, as a slice of bytes, we loop over each byte in turn, seeing if it's possible to increment it without exceeding the maximum value that a byte can hold (255).

If it's possible to increment the byte without overflow, then we do just that, and return the result. With the test case that we saw earlier, for example, `{0, 0, 0}`, it *is* possible to increment the first byte, so that's our result: `Next` should return `{1, 0, 0}`.

On the other hand, if the byte is *already* 255, then its next value should be zero, and we need to "carry" 1 to the next byte. So we zero this byte and move on to the next. The second test case `{255, 0, 0}` is an example of this. We rotate the first byte around to zero, like a car odometer, and the next digit clocks up from zero to one, with the result `{0, 1, 0}`.

But suppose all the bytes are already 255, as in the "overflow" test. In that case, we try to increment the first, rotating it to zero, and carry one. Then we try to increment the second, rotating it to zero and carrying one. We try to increment the third, rotating it to zero—but now we've reached the end of the loop. There's no next byte to carry the 1 to.

In this case, we can't represent the next value within the bytes we have available (the value *overflows* our storage capacity) so we return an error indicating "no more keys".

Checking our key guesses

That's part 1 of the problem solved, so let's turn to part 2: checking all keys against the crib. We've already written the test:

```
func TestCrack(t *testing.T) {
    t.Parallel()
    plaintext := []byte("This message is exactly 32 bytes")
    ciphertext := []byte("Uijt!nfttbhf!jt!fybdumz!43!czuft")
    want := testKey
    got, err := shift.Crack(ciphertext, plaintext)
    if err != nil {
        t.Fatal(err)
    }
    if !bytes.Equal(want, got) {
        t.Fatalf("want %q, got %q", want, got)
    }
}
```

We're using the (now) familiar plaintext enciphered with the "010101..." test key. `Crack` should be able to iterate through all the possible 32-byte keys until it finds the test key, at which point the deciphering will succeed in matching the crib.

The soul of a new cracker

So let's write the new version of `Crack` and see how it does. Here's the code:

```
func Crack(ciphertext, crib []byte) (key []byte, err error) {
    plaintext := make([]byte, len(crib))
    key = make([]byte, BlockSize)
    for {
        block, err := NewCipher(key)
        if err != nil {
            panic(err)
        }
        if bytes.Equal(block, ciphertext[:BlockSize]) {
            return key, nil
        }
        key = rotate(key)
    }
}
```

```
    }

    block.Decrypt(plaintext, ciphertext[:len(crib)])
    if bytes.Equal(crib, plaintext) {
        return key, nil
    }
    key, err = Next(key)
    if err != nil {
        return nil, errors.New("no key found")
    }
}

}
```

Very simple. We start with a key that's all zeroes (thanks to `make`). If that happens to decipher the input, then we're done (gosh, that was lucky!). If not, though, we get the next key with `Next`, and keep repeating this process forever, or more precisely until `Next` returns an error indicating that there are no more keys.

We apologise for the delay

Let's run the test:

```
go test
```

Hmm. Nothing's happening. Let's give it a bit longer and see if it eventually terminates.

Quite a while later:

```
panic: test timed out after 10m
running tests:
    TestCrack (10m)
```

Oh dear. A timeout doesn't necessarily mean a failure, but it does mean that we need to check our expectations. Assuming `crack` is working properly, how long would we expect it to take to find the key for this test case?

Benchmarking key cracking

Let's write a *benchmark* to find out. As you probably know, Go's testing package has benchmarking built in, so this is a nice opportunity to use it. All we need to do is write some function whose name begins with `Benchmark`. For example:

```
func BenchmarkCrack(b *testing.B) {
    plaintext := []byte("This message is exactly 32 bytes")
    ciphertext := []byte("Uiis message is exactly 32 bytes")
    b.ResetTimer()
    for range b.N {
        _, _ = shift.Crack(ciphertext, plaintext)
    }
}
```

Within that benchmark function, we're expected to call the function under test (`crack`, in this case) many times inside a loop. How many times? Well, that's determined by the benchmark machinery.

It makes sense that if we want an accurate measurement of how long `crack` takes, we should call it multiple times and take the average. Looping up to `b.N` takes care of this for us: the benchmark will be executed with different values of `b.N`, and we'll get some hopefully meaningful stats.

Pick an easier problem

Notice that in this benchmark we're not asking `Crack` to find the key that we used in the previous test (`01 01 01...`). We already know that this test times out, so maybe it's just taking too long to find this key. To get a rough

idea how long that would actually take, we'll try cracking a shorter key first, and then extrapolate from that result.

We know that `crack` will try keys in the following order: first, `00 00 00...`, then `01 00 00...`, `02 00 00...` and so on. So we've set up a ciphertext which enciphers the plaintext using the key `01 01 00...`, where the remaining bytes are all zero.

That should be relatively quick to find. The first 256 guesses will be wrong, since they will all have the second byte equal to 0, where we know it needs to be 1. But as soon as the first byte rotates around back to zero and we carry one to the next byte, we will be at the key `00 01 00...`, and then the next attempt will be the correct key.

That's 257 tries in all, so now let's see how long that takes. First, we'll need to comment out `TestCrack`, because running a benchmark *also* runs all tests, and we can't do that at the moment because of the timeout problem.

Understanding benchmark output

With the `crack` test disabled for now, we can go ahead and run:

```
go test -bench .

goos: darwin
goarch: amd64
pkg: github.com/bitfield/shift
cpu: Intel(R) Core(TM) i7-3615QM CPU @ 2.30GHz
BenchmarkCrack-8          48979           24456 ns/op
```

We can ignore the preamble, as it's just telling us some metadata about the system we're running the benchmark on: an 8-way Core i7 with macOS. The interesting line here is the last one:

BenchmarkCrack-8	48979	24456 ns/op
------------------	-------	-------------

The “48979” is the number of times the benchmark loop was executed (that’s the `b.N` value). The “24456 ns/op” is the number of “nanoseconds per operation”, that is, the mean average execution time for crack was 24,000 nanoseconds and change.

This works out to 90-odd nanoseconds per key attempt. Let’s see if we can use that number to predict how long it’ll take to find the key `01 01 01 00 . . .`, in other words, the equivalent of a three-byte key as opposed to a two-byte key.

Cranking up the difficulty

We know from our prior study of keyspaces that this should take roughly 256 times longer, since each additional byte makes the keyspace 256 times bigger. Let’s find out. First, we’ll adjust the input ciphertext so that the first three bytes are offset by 1:

```
ciphertext := []byte("Uijs message is exactly 32 bytes")
```

And try the benchmark again:

```
go test -bench .
```

```
...
BenchmarkCrack-8           192          6193617 ns/op
```

That’s roughly 6 microseconds, or 253 times longer than last time, which isn’t too far off our extrapolated guess. So finding *four* bytes should take about 256 times longer again: 1.6 billion nanoseconds, or about a second and a half.

Here’s the ciphertext:

```
ciphertext := []byte("Uijt message is exactly 32 bytes")
```

And here's the result:

```
go test -bench .
```

```
... BenchmarkCrack-8 1 1593668500 ns/op
```

Not bad! Very close to our predicted figure. It looks like we have a relatively stable way to predict key cracking times. Let's use it to estimate the time to find a 16-byte key, for example.

Ballparking a realistic crack time

That's 12 bytes longer than the key we just cracked, so, since each additional byte multiplies the crack time by 256, we need to multiply our reference time by 256^{12} . That's an awfully big number:

18,446,744,070,000,000,000

Well, that's roughly how many seconds it will take to crack a 16-byte key. Dividing into that to get the time in years gives us something like 584 *trillion* years.

That's already a little longer than we're probably prepared to wait for a test to run, but remember it gives us only 16 bytes. Cracking the full 32-byte key would take another 256^{16} times longer. Check my math, but that works out to many trillion quadrillion times the current age of the universe.

Yeah, no. We might have been too ambitious with this test. Bear in mind we're trying to find a *known* key, and one that's relatively close to the

beginning of the keyspace. Even that would take a brain-meltingly long time, we find.

It *does* give us a nice warm feeling of security about the keyspace of our 256-bit keys, which is so staggeringly gigantic that it effectively renders brute-forcing a waste of time. Instead, it would always be more efficient for Eve to spend, say, the first hundred million years or so closely studying the cipher scheme, looking for flaws that could save her a billion years of key-guessing.

And it makes sense that if *Eve* can't crack our keys, in practice, within a reasonable time, then neither can we do so in a test. So how can we verify that the cracking code actually works, without waiting for it to find real keys?

A test we might live to see pass

Well, on reflection, we really don't need to *test* the `crack` function against a key that takes so long to reach by sequential enumeration. We only need to know two things: first, that it will correctly enumerate every key, given enough time, and second, that when it *finds* the correct key, it will stop.

Fortunately, we know the order in which `crack` will guess keys, so we can save some testing time by deliberately choosing a key which occurs fairly early in that sequence. For example, “01 01 01 00...” has its first three bytes set to 1, and the rest are zeroes.

We already know from our benchmarking that we can find this key within about 6 microseconds, and that sounds fine. We never want our test suite as a whole to take more than a handful of seconds to run, or we simply wouldn't bother running it very often.

So let's update the test for `crack` to give it an input enciphered with this relatively “easy” key:

```

func TestCrack(t *testing.T) {
    t.Parallel()
    plaintext := []byte("This message is exactly 32 bytes")
    ciphertext := []byte("Uijs message is exactly 32 bytes")
    want := append([]byte{1, 1, 1}, bytes.Repeat([]byte{0},
29)...)
    got, err := shift.Crack(ciphertext, plaintext)
    if err != nil {
        t.Fatal(err)
    }
    if !bytes.Equal(want, got) {
        t.Fatalf("want %x, got %x", want, got)
    }
}

```

We already know from our benchmarking that this takes about 6 microseconds (at least on my machine; it's probably faster on yours).

Updating the `crack` tool

We're back to all passing tests, which means we have a working `crack` function. Next, let's update our `crack` command-line tool to take account of the new cipher mode and padding changes.

GOAL: Update `crack` to use `NewDecrypter`.

Putting it all together

HINT: It's been a while since we looked at the `crack` tool, but it doesn't really need that many changes. We can think of it as being essentially the same as `decipher`, with the difference that instead of taking the key as a command-line argument, we call `crack` to guess it.

So you can put this together by taking the existing first part of `crack` (guessing the key), and the second part of `decipher` (deciphering the

message, given the key). See what you can do!

SOLUTION: Here's a version that should do the job:

```
func main() {
    crib := flag.String("crib", "", "crib text")
    flag.Parse()
    if *crib == "" {
        fmt.Fprintln(os.Stderr, "Please specify a crib text with
-crib")
        os.Exit(1)
    }
    ciphertext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    key, err := shift.Crack(ciphertext, []byte(*crib))
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := shift.NewCipher(key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    mode := shift.NewDecrypter(block)
    plaintext := make([]byte, len(ciphertext))
    mode.CryptBlocks(plaintext, ciphertext)
    plaintext = shift.Unpad(plaintext, mode.BlockSize())
```

```
        os.Stdout.Write(plaintext)  
    }  
}
```

Will it crack?

If this works, we should be able to encipher our sample text with the key 01 01 01 00..., let's say, and fairly quickly recover it with `crack`. Here goes:

Let's check the ciphertext:

If tiger appears at its own plebvre. When we become very silenu
...

Okay, not the *most* heavily-obsured message, but we're deliberately using a super-weak key, because we don't have a quadrillion years to wait around for the next command to run:

```
go run ./cmd/crack -crib The <enciphered.bin
```

The tiger appears at its own pleasure. When we become very silent
...
...

Great. Mischief managed!

Parallelisation

During our discussions of brute-force cipher cracking, it might have occurred to you to ask “Couldn’t we use parallel processing here?” In other words, if we can crack a given key in a given time with a single CPU core, wouldn’t we be able to crack it faster using multiple cores?

Certainly we would. As we saw in an earlier chapter, the hard limit on Eve’s ability to crack keys is not really time, but *energy*. Cracking a given key requires, on average, a fixed amount of computation, and thus energy. If she throws more hardware at the problem, she just burns that energy faster. Her electricity bill, as we’ve seen, will work out the same either way.

On the other hand, we have to assume that Eve has the means to put a lot of energy into this cracking operation. (Maybe she *is* the electricity company. I’ve always thought there was something shady about them.)

There are physical limits to how fast computation can happen, though, even with parallel processing. Just for fun, let’s take a moment to look at these.

Big computers are too slow

Suppose Eve wants to apply N cores to the task of cracking Alice’s key. Since each core occupies some amount of space, the average distance between cores increases with N . Cores can’t communicate with each other faster than the speed of light, and *some* amount of inter-core communication is required even for parallel processing.

This implies that there’s some upper limit for N , above which Eve’s galactic megacomputer actually gets slower, not faster. The reason is that there are so many cores that it takes too long for them to talk to each other.

Dense computers become black holes

“No problem,” says Eve, “I’ll just make the cores smaller, or more efficient.” By doing more computation in a given volume of space, she can reduce the light-travel time between cores. But, again, there’s a limit to this. The more mass or energy you try to squash into a small space, the more

strongly it curves the fabric of spacetime, according to Einstein's equations of general relativity.

When the computer (or anything else) reaches a certain critical density, the spacetime in its vicinity becomes so strongly curved that nothing can escape it: no matter, no light, and no useful information. In other words, Eve has created a new black hole.

This would be unfortunate. Yes, Eve can build an arbitrarily powerful computer if she wants to, and use it to crack Alice's key in an arbitrarily short time. But she can never see the answer! Or, at least, she can see it by entering the black hole herself, but then it's no use to her, since she can never again interact with the rest of the universe, including Alice.

So physics gets Eve both coming and going. If her computer is too big, it's slow, because its internal bandwidth is limited by the speed of light. If the computer is too small, it becomes permanently sealed off from the universe inside a black hole, so its results are inaccessible.

Parallel cracking is still useful

These hard limits on key-cracking are unlikely to affect any Earthly Eve, of course, at least for some centuries to come. Returning to the realm of the practical, though, wouldn't throwing at least a *few* extra computers at the problem help Eve a lot? It seems like it should:

A brute-force attack is tailor-made for parallel processors.
—Bruce Schneier, “[Applied Cryptography](#)”

Indeed, if Alice uses short keys, Eve can crack them very quickly, as we've done ourselves, and we could have cracked them in even less time by making our `crack` function concurrent. For example, if we have N cores available, we can divide the search space into N pieces and assign each one to a goroutine running `crack` in parallel.

But we won't implement it here

So, is it worth us modifying `crack` to work in parallel? Not really. As we've discussed in previous chapters, Alice can increase the key-cracking time a lot faster by adding more key bits than Eve can decrease it by adding more cores. Eve would be bringing a linear knife to an exponential gunfight, so to speak.

And, as we've also found by experiment, the expected time to crack *strong* keys (256 bits or longer) is so vast that, really, dividing it by 8 doesn't make much difference. Dividing it by *any* feasible number of cores doesn't make enough impact to be worthwhile.

If Eve, even by commandeering every computer on Earth, can't reduce the expected crack time of Alice's key to less than a hundred trillion years, then what's the point, really? It's hard to imagine any secret information that would still be valuable in even a hundred years, never mind a hundred trillion.

Strong keys can't be cracked

Of course, people do use weak keys, through accident, laziness, or ignorance, and Eve will crack them as fast as she possibly can, in parallel, with the computing resources she has. But that's of limited interest to us, since we won't be *using* weak keys. We know how to make our keys so strong that attempting to brute-force them is a waste of time.

Perhaps a good way to make the point would be that strong keys take so long to crack that parallel processing doesn't help, while weak keys can usually be cracked so quickly that parallel processing isn't necessary. Either way, we needn't worry about it for our cipher system.

If you'd like to try writing a parallel version of `crack`, though, just for fun, of course you're very welcome to have a go. For now, let's move on to thinking about another thing that's essential to strong keys: *entropy*.

12. Entropy

Random numbers should not be generated with a method chosen at random.

—Donald Knuth, “[The Art of Computer Programming, Vol 2: Seminumerical Algorithms](#)”



(photo by [Dean Hochman](#), licensed under [Creative Commons](#))

So now we know a bit about strong keys. During our various exciting adventures in keyspace, we've learned how the size of a key affects the security of the messages it enciphers.

And we also learned that size isn't everything. Or rather, it's not just the length of the key that's important, but how many *possible* keys there are. Let's talk a little more about that.

Information

In an earlier chapter we saw that any key can, in principle, be generated by some repeated process with a binary outcome: for example, tossing a coin.

This in turn implies that any key can be *represented* by a binary number: that is, a string of bits. Each bit could be thought of as the result of a coin-toss: 0 or 1, heads or tails.

But we also know that not all possible bit-strings make equally good encryption keys. A key of all zeroes results in no encryption at all, for example, and a key that's *mostly* zeroes doesn't encrypt the plaintext very much.

A key of all ones would be equally ineffective, and so would any key with an obvious, predictable pattern in its bits. For example, 0101...

So there's something else that makes a key more or less secure, beyond its basic information content given by the number of bits. We could think of this extra factor as how much *new* information it contains. Equivalently, we could describe it as how much Eve doesn't already know about the key.

Entropy: a measure of our ignorance

This quantity is sometimes referred to as *entropy*, which is a very good word to drop into conversation if you want to sound knowledgeable.

Indeed, it's one of those words that means lots of different things to different people. Most of them would agree, though, that it has something to do with information, particularly *unknown* information.

In the context of cryptography, we can be quite exact about what we mean by the “entropy” of a given key or message. It's the minimum number of bits required to write down the information it contains.

Mathematically:

In general, the entropy of a message measured in bits is $\log_2 N$, in which N is the number of possible meanings. This assumes that each meaning is equally likely.

—Bruce Schneier, “[Applied Cryptography](#)”

In other words, if there are N possible messages, then the entropy of any particular choice of message is the number of bits required to uniquely express which of those possibilities you chose. Equivalently, it's the number of coin tosses required to generate that message.

Note that this needn't be the same as the number of bits *in* the message itself. Some of those bits might be *redundant*: that is, they don't add anything to the unique information content of the message.

This is true of English texts as well. You might have seen the old billboard adverts for a secretarial college which said something like this:

F U CN RD THIS, U CAN BCM A SEC & GT A GD JB W HI PA

And you *can* read it, right? That's the point: English spelling is redundant enough that we can remove many of the letters without impairing the content of the message.

That's a good thing for us in general, but it's not so good in the context of cryptography. If a key contains redundant bits, for example, then by definition those bits don't contribute anything to the security of the key. We should subtract them from the effective length of the key for cryptographic purposes.

A 256-bit key that contains 250 redundant bits is really no more secure than a 6-bit key: it just *looks* as though it is. So we would say that, even though the key is 256 bits long, its *entropy*—the number of useful bits—is only 6.

Entropy of digital messages

This gives us a powerful way of calculating just how hard Eve's task of brute-force guessing will be, for a given key or message. It even applies to things that are non-textual in nature.

For example, suppose Alice wants to send a photo privately to Bob. We won't be so intrusive as to ask what's in the photo, but we would like to know how much entropy there might be in Alice's message.

If Eve wants to peep on Alice and Bob's encrypted communications, this is equivalent to asking how many possible plaintexts she has to consider if she doesn't know the cipher key. So, can we work it out?

Well, the *raw* sensor data from Alice's camera might be something like 100MiB per frame, but as we now know, the entropy will probably be somewhat less. Image data usually contains some redundancy.

Pictures often contain a lot of adjacent pixels of the same colour, so Alice may be able to *compress* the image file without losing any information. For example, *run-length encoding* specifies that a given pixel should be repeated a number of times. If the image contains many long runs of identical pixels, it can be compressed a lot, but the original image can still be recovered exactly.

Compression

And this is another way of thinking about entropy: it's what's left after you've applied as much of this *lossless* compression as possible.

You can still go ahead and apply *lossy* compression too, of course, but unlike lossless compression, that *does* reduce the entropy of the message. That's often okay for data such as images or movies, since what's lost in lossy compression is usually information we can manage fairly well without.

For example, many image compression schemes sacrifice some of the *dynamic range* of the image (the maximum difference in brightness between dark and light areas). Or they give up some of the original *colour depth*: the range of available colours for each pixel. Digital sensors can discriminate finer shades of colour than the human eye, so we can lose a surprising number of bits per pixel without affecting the image too noticeably.

The same principle applies to compressing music, voices, or other sounds. Humans don't hear frequencies above about 20KHz very well, so we can remove most such frequencies from a digital music file without making it

sound terrible. Indeed, the worse the sound quality you're willing to accept, the more you can compress the audio data.

Effective lossy compression, then, depends on knowing something about the *domain* of the data you're dealing with. If it's still images or video, you can take advantage of the characteristics of the human eye to eliminate some less important information. If it's audio, you can do the same by knowing about the differential frequency response of the ear, and so on.

Enciphered data

So what about compressing *enciphered* data? In principle, that shouldn't be possible. Every bit of the ciphertext should be essential for reconstructing the plaintext. Any redundant bits, by definition, can be removed without affecting the decipherability of the message.

Redundancy is great when we *want* our message to be easily understood, as in the billboard example, or in circumstances of noisy or imperfect transmission. But when we want to *conceal* the message, the same reasoning implies that there should be little or no redundancy in the ciphertext.

It shouldn't be possible to compress the ciphertext by much, then, because there should be no redundant bits to remove.

Identifying ciphertexts

This is something it's useful for Eve to know, too. As she pores over the stream of intercepted binary data from Alice and Bob's communication channel, her first challenge is simply to identify which of the messages are actually enciphered.

Many of them won't be, so how can Eve tell if a given sequence of bits represents ciphertext or not?

That's easy: she can try to compress it! If this doesn't appreciably reduce its size, then either it's already compressed, or it's ciphertext (or both).

Complexity

This idea of stripping out redundant information gives us another interesting way of thinking about the entropy of a key or message.

Suppose Alice wants to send Bob a list of all the positive, even integers less than a million. What's the most efficient way she could do that?

Describing sequences

The *obvious*, but inefficient, way would be to simply write them all out:

2, 4, 6, 8...

But that would be a long message: many millions of bytes. And we instinctively recognise that this long-windedness isn't necessary.

Because the concept of “positive and even numbers” is familiar to most people, Alice could use it to shorten her message:

Dear Bob,

I hope this letter finds you well. I am writing to inform you of the positive, even integers less than one million.

I beg to remain,
Your obedient servant,
Alice

Perhaps this message could be made even shorter, at the risk of being slightly less courteous, but you see the point, I'm sure.

Instead of laboriously writing out every even number, all Alice has to do to convey that information to Bob is to unambiguously *describe* that set of numbers instead.

A generating algorithm

In a sense, what she's sending him is not the list of numbers itself, but a kind of recipe for producing it. Or, as a computer scientist would say, an *algorithm*.

Although she didn't say so explicitly, Alice is telling Bob that if he writes out all the positive, even integers less than one million, he will have the list she wants to communicate to him. Because the numbers form an easily-described sequence, it's much more efficient to send the *description* than it is to send the sequence itself.

To eliminate any ambiguities or uncertainties, Alice could make this description so explicit that even a machine could understand it. This would naturally take the form of a computer program.

It happens that we know a very good programming language she could use, so we might suggest to her something like the following:

```
package main

import "fmt"

func main() {
    for i := 1; i < 1_000_000; i++ {
        if i%2 == 0 {
            fmt.Println(i, ", ")
        }
    }
}
```

Bob doesn't need to know how any of this works, or even to understand Go code. He can simply copy and paste it into the [Go Playground](#), or a text file, and run the program. The result is, of course, the message that Alice intended to communicate:

2, 4, 6, 8...

The shortest program that describes a sequence

Now, although this program is a little longer than the English words “the positive, even integers less than one million”, it *is* very precise. It’s also a great deal shorter than writing out all the numbers in full. In fact, it takes up just 123 bytes, or 984 bits.

So Alice can send Bob just 984 bits of information, and contained within them is everything necessary to reconstruct the original message (except for a Go compiler, of course).

This, then, is a very effective kind of lossless compression. Instead of sending the data itself, send a *program* that generates the data.

This is a good way to discover how much entropy is contained in a message, isn’t it? For a given string of bits, let’s say, we can ask “What is the shortest program that generates exactly these bits?”

Kolmogorov complexity

This measure is called the *Kolmogorov complexity*, or *algorithmic entropy* of the message. When we have to pay by the bit to transmit some message s , we want its Kolmogorov complexity $K(s)$ to be as low as possible, to minimise our phone bill.

Some apparently very long messages—such as the sequence of integers in our example—turn out to have a very low K . This is equivalent to saying that they can be losslessly compressed into a much smaller message.

Indeed, many *infinite* sequences are like this, such as:

- all the even integers
- all the integers
- the digits of π

and so on. All these sequences can be completely described by very concise programs, making them very compressible (infinitely so, indeed).

How does this low entropy relate to randomness? While the integer sequences don't look random, the digits of π certainly do:

3.14159265358979323846264338327950288419716 . . .

There's *apparent* complexity here, but in fact the sequence is completely described by a very regular-looking formula:

$$\$ \$ \frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \$ \$$$

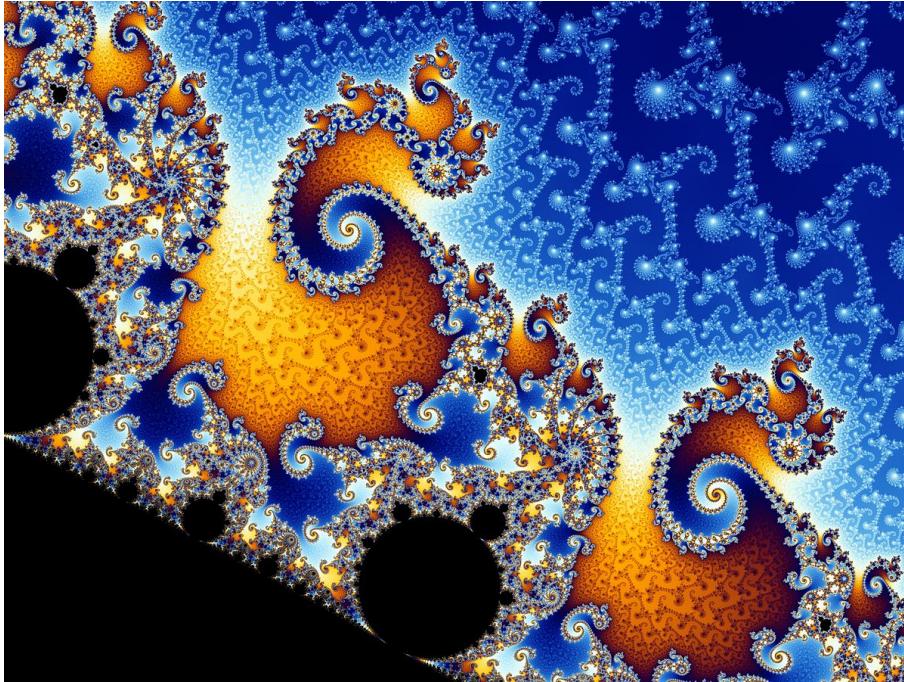
And I don't even really need to send you the formula. I can just say "pi", and you'll know what I mean. That's enough information for you to extract any specific set of digits you need, given enough time and computer power.

This really highlights the difference between information and entropy, doesn't it? π would take an infinite number of bits to write out in full, but I can completely specify the values of all those bits by a single Greek letter, which takes 32 bits to encode in a Go program.

Therefore, in this particular scheme the entropy of π is just 32 bits. It wouldn't make a good encryption key!

Complexity from simplicity

Similarly, consider the famous Mandelbrot set:



(image by [Wolfgang Beyer](#))

This looks complex, but it's generated from this simple formula:

$$z \rightarrow z^2 + c$$

According to Kolmogorov, a system is only as complex as its simplest description, and this description is pretty simple!

That's the opposite of what we want for a cryptographic key, though. A key that *looks* long and complex, but that is completely described by a short and simple rule, is a gift for Eve, and thus a disaster for Alice. Such a low-entropy key is insecure.

The security of an algorithm rests in the key. If you're using a cryptographically weak process to generate keys, then your whole system is weak.

—Bruce Schneier, [“Applied Cryptography”](#)

Security

Not all complexity is illusory, though. Some complex-looking things really *are* complex: they have a very high K , meaning that they contain a lot of entropy.

Randomness is high entropy

For example, consider these lists of numbers: the diameters of each sand grain on every beach in the world, or the masses of each star in the galaxy. These are high- K messages: the shortest program to generate them would be no shorter than the messages themselves.

This gives us another clue about what *makes* random sequences random: they contain a lot of entropy. Equivalently, they have high Kolmogorov complexity.

To say that you can't write a short program to generate a given sequence is the same as saying that the sequence is unpredictable: that is, random. While the digits of π look random, they aren't, because they form a completely predictable sequence.

Key generation

The reason this is significant for things like passwords and keys is that Eve's chance of guessing them is directly related to the amount of entropy they contain.

Even if Alice's password is a million bits long, it won't be very secure if most of those bits are redundant. For example, maybe the whole bit-string can be generated by a program no more than 128 bits long. This is the same as saying Alice's key contains only 128 bits of entropy, even though it looks much longer.

To break Alice's key, Eve only has to search the space of 128-bit programs, not the space of 1-million-bit numbers. Alice would have done better to simply toss a coin a million times (or even just 129 times) to generate her key.

Sure, she *might* end up by pure chance with a very low- K number, such as all zeroes, or alternating ones and zeroes. But such numbers are a vanishingly small subset of the $2^{1,000,000}$ possible keys. Most of these, statistically, won't form easily-described predictable sequences.

Choosing random bits, then, is a good way to create secure keys, because by definition they have a lot of entropy.

Pragmatic non-determinism

But how can we choose bits randomly with a *program*? A computer can't toss a coin, after all. Or can it?

I mean, the result of a coin toss isn't unpredictable *in principle*. If you knew the exact shape and mass distribution of the coin, its initial velocity and spin, the location and momentum of every air molecule nearby, and so on, you could work out whether it's going to land heads or tails.

But, in practice, you don't, so you can't.

The reason a coin toss is “acceptably” random for most human purposes is that even though the result is deterministic, actually *determining* it is more trouble than anyone's usually willing to go to. This is because it's the result of a *chaotic* system. “Chaos”, in the mathematical sense, means that the way such a system evolves over time is highly sensitive to initial conditions.

That's true of many processes in the physical world, in fact. Reality has a lot of decimal places. It doesn't take many interactions between physical things for the results of future measurements to become, in practice, impossible to calculate with any useful level of accuracy. That's why our ability to predict the weather falls off so sharply with time.

A coin toss is a good example, because a small difference in the force with which the coin is tossed could change a heads outcome to a tails, or vice versa. Similarly, tiny air movements or slight differences in the weight distribution of the coin can have a large effect on the result.

In the next chapter, we'll look at some practical ways in which chaotic physical processes can be used as a source of entropy. One fun, if slightly *impractical* way, though, is to use a lava lamp, as seen in the photo at the beginning of this chapter.

Cloudflare's office in San Francisco features a "[Wall of Entropy](#)", using about 100 lava lamps. A digital camera takes a picture of the lamps at intervals, and the data from these pixels is used as input entropy for generating cryptographic keys. So, while the wall of lava lamps makes a nice display for visitors, it's not just a conversation piece.

A little entropy goes a long way

The ability to generate random numbers is important for cryptography, as we've seen, but it's also useful in other spheres of life. For example, poker players need to be able to randomise their behaviour, to avoid their opponents learning too much about their play.

Poker, like cryptography, is a game of information. The more regular and predictable your play, the easier it is for opponents to beat you.

To inject randomness into their game, some players use their analogue wristwatches as a simple random generator. For example, if the second hand is between 0 and 30, they'll raise on a middling hand, and if it's between 30 and 60, they'll call.

This wouldn't work in every situation, of course: if you needed to generate random numbers more than about once a minute, the stream of "bits" would start to show obvious regularities. And if your opponents knew your exact scheme, they could use their own watches to predict what you'll do. But even a little randomness—even a small amount of entropy—can give you an important edge.

Being unpredictable

The same idea applies to games like "rock, paper, scissors". Although it doesn't seem like it, this is very much a game of skill. It takes skill to be

able to *avoid* predictability in your moves, and to *spot* regularities in your opponent's.

Being unpredictable, indeed, is the hard part: our little meat brains just aren't wired for it. No matter how hard you try, your attempts to generate entropy won't be very successful. Without realising it, you'll fall into relatively predictable patterns, and a more experienced player will easily beat you.

LECTER: *Clarice, doesn't this random scattering of sites seem desperately random, like the elaborations of a bad liar?*
—[“The Silence of the Lambs”](#)

The more we try to be random, the more we give ourselves away. Being able to produce sufficiently random numbers, then, is a key part of cryptography, and we'll explore this further in the next chapter.

13. Randomness

The meat of the book is the “Table of Random Digits”... The table goes on for 400 pages and, except for a particularly racy section on page 283 which reads “69696,” makes for a boring read.

—Bruce Schneier, [“Applied Cryptography”](#)



In the previous chapter we saw that choosing good cryptographic keys—or making any other kind of choice that opponents mustn’t be able to predict—depends on our ability to behave randomly. Or, at least, to behave in a way that *looks* random enough to fool those who are trying to out-guess us, as in games such as poker and rock-paper-scissors.

Pseudo-random generation

Computer games also often need random-looking behaviour, to make the player’s experience a little different each time. But, like the Mandelbrot set, random number generators for games only need to *appear* complex. In practice, for speed and ease of implementation, they can be very simple indeed.

Randomness in games

Games like Elite need a steady stream of random numbers. They are used all over the place to add an element of chance to gameplay, whether it's in the main flight loop when deciding whether or not to spawn an angry Thargoid in the depths of space, or on arrival in a new system where the market prices have a random element mixed into the procedural generation, so they are never exactly the same.

—Mark Moxon, “[Generating Random Numbers: Elite on the BBC Micro](#)”

The legendary space game [Elite](#) was first implemented on the [BBC Micro](#), a decent home computer in 1984, but a tiny machine by today’s standards. It had a single 8-bit CPU clocked at 2MHz, and 32K of memory. Into this, the authors squeezed a real-time 3D space combat game with dozens of ship types, across eight galaxies containing over two thousand unique star systems to explore.

The Fibonacci sequence

This masterpiece of elegance and efficiency required many clever tricks to fit it into the BBC’s tiny memory, and one of them was an ingenious little “random” number generator in just [12 machine instructions](#).

The idea is based on the Fibonacci sequence, which we’ve met already as one of our “regular” sequences:

0, 1, 1, 2, 3, 5, 8...

Each number, as I’m sure you’ve spotted, is the sum of the previous two. We need two “seed” numbers to start things off, so the actual sequence produced depends on which two numbers you pick.

If you pick zero and one, you get the familiar sequence above. But different seeds will give us a different sequence. For example, using seeds of 12 and 43, we get:

12, 43, 55, 98, 153, 251, 404...

This looks a bit random-ish, except that the numbers are always increasing. We can solve that quite easily using our old friend modular reduction. For example, let's take the residue of each number mod 10 (or, in plainer language, take its last digit):

2, 3, 5, 8, 3, 1, 4...

A simple random generator

This is already perfectly good enough for a simple game like “guess the number I’m thinking of between 0 and 9”. If Alice is playing this game with Eve, it doesn’t matter if Eve knows that Alice is using the Fibonacci sequence, because she’s still missing one crucial piece of information: the *seeds* that Alice chose.

Alice can get these two numbers, as I did, by looking at the clock, or throwing dice, or using any other decent source of randomness. The point is that she doesn’t have to *keep* doing that: once she’s generated the seeds, she can just use a very simple algorithm (addition mod N) to produce as long a random-looking sequence as she wants.

A Fibonacci generator in Go

Here’s a Go program that implements this trick:

```
package main

import (
    "fmt"
    "time"
)

var (
    seed1 = 0
```

```

    seed2 = 1
}

func main() {
    for range time.Tick(time.Second) {
        number := seq()
        fmt.Println(number, ", ")
    }
}

func seq() int {
    tmp := seed1 + seed2
    seed1 = seed2
    seed2 = tmp
    return tmp % 10
}

```

[\(Listing random/1\)](#)

Each call to `seq` generates the next number in the sequence, starting from the two configured seeds. And here's the output:

`1, 2, 3, 5, 8, 3, 1, 4...`

Repeatability

We have a random-seeming sequence, even though it's generated by what's clearly a deterministic program: every time we run it, we'll see the exact same numbers. While this is a disadvantage in cryptography (because if Eve gets hold of Alice's key-generating program, she'll be able to generate the same keys), such repeatability can be desirable in games.

For example, when you destroy an enemy ship in Elite, the random generator is used to draw the explosion particles in different locations on

the screen. For the next frame of the animation, those same particles must be overdrawn, to erase them.

But storing and retrieving the co-ordinates of every particle would take up too much memory, so instead the game simply re-runs the random generator with the same seeds, thus producing the same co-ordinates. The results look random enough to satisfy players, but the repeatability gives good graphical performance.

Seeding the RNG

Even in a game, though, it might start to become a bit obvious if the *seeds* were always the same every time you played it. The same enemy would appear at the same point, attack at the same moment, and so on. Elite solves this by grabbing information from the gameplay to seed its random generator (for example, the current screen *x*-co-ordinate of the nearest planet).

We could use something like the poker player’s wristwatch trick to achieve the same goal. For example, suppose that when the program starts, it chooses initial values for the two seeds based on the current time. We could just take the current “seconds” value of the system time, like this:

```
var (
    seed1 = time.Now().Second()
    seed2 = time.Now().Second()
)
```

([Listing random/2](#))

Even though these are overwhelmingly likely to be the same value, it doesn’t matter. The sequence produced is fine:

4, 1, 5, 6, 1, 7, 8, 5, 3, 8...

And we'll run it again to check that we get something different:

```
2, 3, 5, 8, 3, 1, 4, 5, 9, 4
```

Security problems

This is certainly good enough for games, and other non-cryptographic applications. Indeed, it's more or less the way that Go's standard `math/rand` package generates "random" numbers: a deterministic algorithm, seeded with the current time.

Algorithms like this, that produce *apparently* random results but are nonetheless deterministic, are known as *pseudo-random number generators* (PRNGs). The "pseudo" (Greek for "false") is to remind us that such programs can't be used for cryptography, because they produce fatally insecure keys, and we'll very soon find out why.

Periodicity

Let's see what effect a different choice of seeds has on our program's output:

```
var (
    seed1 = 5
    seed2 = 5
)
```

Here's the result:

```
0, 5, 5, 0, 5, 5, 0, 5, 5...
```

Oh dear. This doesn't look nearly so good. We've stumbled upon one of the disadvantages of PRNGs: *periodicity*. In other words, the sequence eventually repeats. If you're unlucky with your choice of seeds, as we were, then the algorithm's period can be very short.

No matter how carefully the seeds are chosen, though, any deterministic sequence will eventually repeat, and that's a fatal flaw of PRNGs when applied to cryptography. Eve doesn't even have to know what the algorithm is, only that if she waits long enough, the digits of Alice's key will cycle back round to where they began.

Distribution

So that's one important reason why the output of deterministic PRNGs like `math/rand` isn't really random: it's repetitive. Another problem is the *statistics* of such sequences. For example, let's see what happens if we choose the seeds 2 and 4 for our program:

```
6, 0, 6, 6, 2, 8, 0, 8, 8, 6, 4
```

It's clear that we'll never get any odd numbers in this sequence. Adding together two even numbers can only ever produce an even result. So, while the program hasn't yet got trapped in an obvious short cycle, it is condemned to wander forever among the even digits, never escaping them.

An apparently random sequence that included no odd numbers wouldn't look very random to most of us. And, thinking about it, that's another quality we intuitively ascribe to random numbers: *uniformity*.

Uniformity

In other words, if we generate 1,000 random digits, we expect that 0 will occur roughly as often as 9, or any other particular value. It would be weird, by contrast, if we saw some specific digit occurring a lot more often than any other, or if even numbers occurred more often than odd ones.

Similarly, a fair coin should produce roughly equal numbers of heads and tails. Of course, over very short runs, we wouldn't be surprised to see a few more of one outcome than the other. But if we toss the coin a few hundred or a few thousand times, the results should be extremely close to 50% heads, 50% tails.

That's not true of our Fibonacci-based generator. We would *like* it to produce 50% odd digits, 50% evens, but it certainly doesn't do that when given even seeds. If we're planning to seed it using something arbitrary like the current time, this would be a disaster waiting to happen.

“Secure enough” random generators

So how do *secure* random number generators work, then? We know they exist (at least, for some value of “secure”) because every time you browse a website or make a credit card transaction online, your communication with the server is encrypted using some randomly-generated key. That *can't* be produced by a simple deterministic PRNG like the one in our example, or you'd soon find your bank account empty.

Environmental noise

The answer is that the random generator does something equivalent to tossing a coin, which as we discussed earlier is so sensitively dependent on initial conditions as to make it impractical for Eve to predict the outcome. And there are many sources of such data in a computer: for example, you could use the intervals between keystrokes or mouse movements, or the differing access times of blocks read from disk.

Indeed, randomness becomes just one of the many useful resources that operating systems provide to applications. “Environmental noise” sources such as keystrokes or network packets help to seed the system *randomness pool*, which uses a high-quality PRNG to maintain a sort of entropy reservoir. Programs that need random bits (for example, cryptographic key generators) can simply read as many as they need from the pool.

The system entropy pool

For example, suppose we need to generate a 256-bit key, so we require 256 bits of entropy. On macOS or Linux, we can get them on the command line by reading directly from the `/dev/random` device, which is connected to the system entropy pool:

```
hexdump -n 32 /dev/random
```

```
00000000 d0 05 b7 68 e5 f7 83 59 bd 2f de 89 c4 8a 98 68  
00000010 c6 f0 a8 6f 6f eb 38 a1 9e b7 9c 23 12 d7 8c a9
```

Similarly, when we use Go’s `crypto/rand` package to generate secure random data, it comes ultimately from this system pool. So, is this good enough? I mean, it’s not *really* random: it’s derived from objective measurements of real-world events, via a deterministic PRNG algorithm. Is this “fake randomness” actually secure?

Security is relative

Well, you better hope so, because your own online security depends on it! And, as we’ve discussed throughout this book, it’s never a case of “secure, or not secure”: it’s all relative. Specifically, the security of a given key or message is relative to the amount of effort an attacker is likely to expend trying to break it.

For example, if Eve wants to spy on Alice’s crypto keys, she could do it by injecting some malicious software into Alice’s computer. It doesn’t matter what the software does: let’s say it hijacks the operating system call that reads from the entropy pool. Every time Alice runs a program to generate her new key, Eve intercepts the “random” bits she read from the operating system.

But if Eve has this level of access to Alice’s computer, she doesn’t even need to bother cracking Alice’s message traffic. She can exfiltrate, read, and perhaps even modify Alice’s private files. When Alice uses her cipher program to encrypt a message to Bob, Eve can read the plaintext as Alice types it in.

So cryptography based on a personal computer’s system entropy pool is reasonably secure *against attackers who don’t have access to that computer*. And, sure, Alice’s keys and ciphers are probably vulnerable even without

that access if Eve is simply willing to spend enough money and time to crack them.

You don't need to outrun the bear

But, as we've also established, the best way for Eve to spend that money is almost certainly not on brute-force attacks. Instead, she should just call up Alice and trick her into giving up the key ("This is, uh, Evelyn, from IT? Yeah, we have a network issue over here. I'm gonna need you to go ahead and change your password to `ADMIN` for me.").

Or she could just wait in the basement parking structure of Alice's office and threaten her with a wrench. Or send Alice an email with a link to a fun meme ("The dog is in a little bag! That's so adorable!") that actually compromises her computer. Or sneak into Bob's house and read Alice's messages after he's deciphered them. Or... you get the point.

It's like the old story about two hikers threatened by an angry bear. They don't need to outrun the bear. Each of them just needs to outrun the other. Similarly, you don't need your keys and ciphers to be literally uncrackable (just as well, since they can't be). You just need them to be hard enough to brute-force that it's never the enemy's best option.

Since, as we've discussed, the security of a key depends critically on the randomness of the bits used to generate it, the same applies to your source of randomness. It doesn't need to be perfectly random, only unpredictable enough that it's not worth your enemy's time to try to predict it. And a decent PRNG seeded by environmental noise is good enough for this.

Keeping secrets from the gods

But, just for fun, could we do better? Is it always just a case of making Eve's brute-forcing so expensive that she resorts to the \$5 wrench instead? Or is there a source of *truly* random bits in the universe? Can we use it to outrun not just Eve, but the bear?

Hardware entropy sources

We talked about “environmental noise” as a source of entropy for operating systems, but there are other kinds of noise. *Thermal noise*, for example, is the electronic interference caused by heat in a conductor. Engineers devote a lot of time and ingenuity to getting rid of it, but they never quite can. The upside of this is that when we *want* some nice, juicy noise, so that we can harvest its entropy, it’s right there on the chip.

Since entropy has become so valuable in computing, it’s no surprise that many manufacturers have started to build such hardware-based *true random number generators* (TRNGs) directly into CPUs, along with machine code instructions to read random values from them.

There are other electronical or optical gizmos based on similar principles, and if you don’t happen to have a CPU with its own hardware RNG, you can buy an outboard one as a USB device.

Defeating “God Emperor Eve”

Even thermal noise RNGs aren’t really *perfectly* secure, though. If Eve could measure the kinetic energy of individual atoms and electrons within the device, she could, in principle, calculate the exact values of its output bits. Sure, that’s totally impractical, but we’re not worrying about what’s practical anymore, just geeking out on physics.

Suppose Eve is an alien with highly advanced technology relative to ours, for example. Maybe she *could* remotely and undetectably manipulate each electron in Alice’s hardware RNG. Is there anything Alice can do to protect herself against such a high-tech-level attacker? Is there a source of random bits whose value is unpredictable even *in principle*?

Surprisingly, the answer is yes. Our best theories of physics say that some quantum events are *truly* random (at least, as far as we know). The details don’t matter for this discussion, but the underlying principle is fundamental: the results of quantum measurements are impossible to predict.

Quantum measurements

For example, measuring the polarization direction of a photon—a particle of light—has a binary outcome: you’ll either measure it as one thing or the other. Let’s call them 0 and 1. You don’t know which it’ll be until you see the result, and in a sense, the photon itself doesn’t know. It exists in a *superposition* of the two states, 0 and 1: both answers are true at the same time.

When you measure it, our best current theories imply that you yourself become part of the superposition. One “version” of you remembers measuring 0, the other remembers measuring 1. Both versions of you are equally “real”, some would say, but others would disagree.

What (almost) everyone does agree on, though, is that you can’t know the result in advance, even in principle. There’s no little hidden switch inside the photon which, if you could only sneak a peek at it, would tell you which polarization you’re going to measure. The answer is always “both, but in different realities”, if you like. Weird, but true.

That means it doesn’t matter how advanced Eve’s alien technology is. Even if Eve is a hyperintelligent pan-dimensional being who can harness the energy of black holes and warp spacetime to travel faster than light, she still can’t predict the measurement of a single quantum bit with better than 50% accuracy. Even gods have their limits, it seems.

A quantum randomness source

Quantum randomness, then, is a perfect source of cryptographic entropy, to the best of our knowledge, and like everything else these days, you can get it on the internet. The nice people at [Australian National University](#) have hooked up their hardware RNG to a server with an API that you can call to get as many random bits as you want.

So, if you want to generate a password that’s random enough to defeat even Hyperintelligent Pan-Dimensional Eve, you can use the [grand](#) Go package to do it:

```
// Sign up here to get your API key:  
// https://quantumnumbers.anu.edu.au/  
rnd := rand.New(qrand.NewSource(qrand.NewReader(apiKey)))  
password := make([]byte, 32)  
for i := range password {  
    password[i] = chars[rand.Intn(len(chars))]  
}  
fmt.Println(string(password))
```

([Listing.password](#))

Generating quantum keys

And it's even easier to generate a 32-byte cipher key; for example, one we could use with our shift cipher:

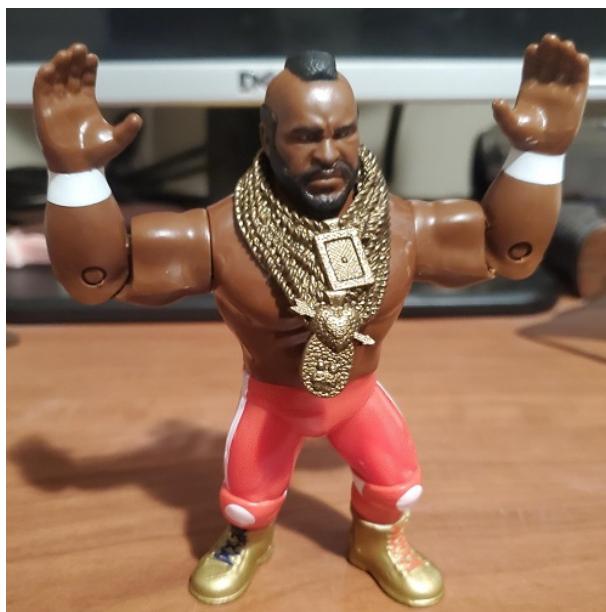
```
q := qrand.NewReader(apiKey)  
buf := make([]byte, 32)  
_, err = q.Read(buf)  
if err != nil {  
    fmt.Fprintln(os.Stderr, err)  
    os.Exit(1)  
}  
fmt.Println(hex.EncodeToString(buf))
```

This is just for fun, though: it wouldn't seriously impede God Emperor Eve, because presumably she could just read the bits off the wire as they arrive over your internet connection, or as they're generated at the ANU data centre. A local source of randomness, such as your system entropy pool, even if it's slightly less random than `qrand` data, is ultimately more secure because it offers fewer opportunities for interception.

14. Chains

The greatest trick the Devil ever pulled was convincing the world he didn't exist.

—["The Usual Suspects"](#)



We've now considerably strengthened our shift cipher by switching to long keys: so much so, indeed, that it would take many times the age of the Universe to crack them by brute force alone. And we've explored some sources of entropy and randomness that will help to ensure that Eve can't predict these keys by attacking a weakness in the generating algorithm.

So let's see how effective our cipher really is at concealing the contents of Alice and Bob's messages. After all, if Eve can still get useful information from an enciphered message *without* cracking the key, that would be a problem. And maybe she can.

Visualising the problem

Let's try enciphering our little devil gopher image using the latest version of the `encipher` tool. But first, a reminder of what he looks like in plaintext form:



Clearly we want the enciphered image to look as different from this as possible. The ideal ciphertext, indeed, would be an image whose pixels are indistinguishable from random noise.

A completely random image

What would that look like? Well, it's easy enough to generate such an image, using a simple Go program:

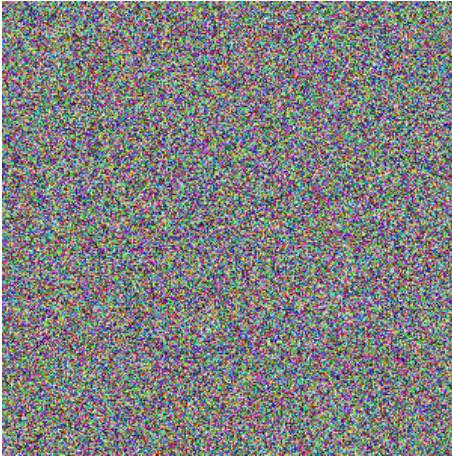
```
package main

import (
    "fmt"
    "image"
    "image/color"
    "image/png"
    "math/rand/v2"
    "os"
)
```

```
func main() {
    width, height := 300, 300
    img := image.NewNRGBA(image.Rect(0, 0, width, height))
    for x := range width {
        for y := range height {
            img.Set(x, y, color.RGBA{
                R: uint8(rand.Intn(256)),
                G: uint8(rand.Intn(256)),
                B: uint8(rand.Intn(256)),
                A: 255,
            })
        }
    }
    f, err := os.Create("random.png")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    defer f.Close()
    err = png.Encode(f, img)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

([Listing randimg/1](#))

For an image of a specified width and height in pixels, we set each pixel to a random colour. Here's the result:



That *really* doesn't look like anything, or rather, it looks like everything! So that's the gold standard, if you like. We should be able to encipher an image in such a way that anyone without the key would have no idea at all what the image represented. It should always just end up looking like the image we just saw: a mess of random, high-entropy noise.

Separating metadata and pixel data

One thing about enciphering a PNG file, as we saw in an earlier chapter, is that it doesn't just scramble the *payload*—the actual pixel data—it also scrambles the header. Without the metadata that identifies the file as a PNG image, we can't easily open the result in an image viewer to see what it looks like.

But there's a way round this. Suppose we take the image file and separate the header part from the payload. Then we can encipher *just* the payload (that is, the pixels). If we reattach the result to the original header, we should end up with a valid image file, but one whose *pixels* are enciphered.

That's a bit tricky to do with PNG or JPG formats, as it happens, but there's another format that makes this splitting and rejoining easier: the Portable PixMap (PPM). It's a now very obsolete image format, and highly inefficient, but it was designed to make image data manipulable as *text* (for example, for sending images in emails).

And that suits us very well, because it has a useful property: the header data occupies the first three lines of the file, while the rest is payload. That makes it easy to separate using standard command-line tools such as `head` and `tail`.

Hidden figures

First, we'll convert our `devil.png` image to PPM format. You can find the result in the example repo as [`devil.ppm`](#).

He still looks exactly the same (but the file is a lot bigger, and I converted transparent pixels to white, because PPM doesn't support transparency):



Next, let's strip the first three lines from this PPM file and save them as `header.txt`:

```
head -n 3 devil.ppm >header.txt
```

We'll be needing those later. The next step is to copy the image data—all *but* the header lines—to another file:

```
tail -n +4 devil.ppm >imgdata.txt
```

We're ready to encipher just the image data. Let's use a key of "all ones", to start with:

And now the payoff. We'll reattach the enciphered image to its header, creating a new, valid PPM file:

```
cat header.txt enciphered.bin >test.ppm
```

Here's the result, opened in an image viewer:



Wow. That's *terrible*. Far from obscuring the original image, the encryption hardly seems to have altered it at all. The background went black, and the eyes went a bit swirly, but that's basically it.

Muddying the waters

If we hoped that Eve wouldn't be able to spot that Alice's message is a picture of a devil gopher, the result is disappointing. Indeed, Eve doesn't need to break our cipher scheme at all, whether by brute-forcing the key, or otherwise. All she needs to know is that the ciphertext contains image data, and she'd be able to reconstruct something like the picture we just saw.

So what's gone wrong? We deliberately chose a "low" key: one that doesn't perturb the plaintext very much. It just adds one to the value of every byte. Maybe that's why it's so shatteringly obvious what the enciphered picture represents.

Let's try the process again, only this time we'll use a randomly-chosen 32-byte key. That should perturb the image data a lot more, and hopefully obscure it more effectively.

In fact, to avoid any worries about whether or not the local computer has a good enough source of randomness, I've used the [qrand](#) tool, which we met in the previous chapter, to generate a key of truly cosmic randomness:

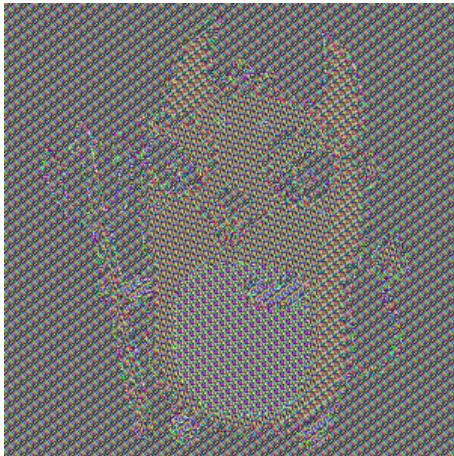
```
qrand 32
```

```
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1
```

You can use any method you like, including tossing a coin 256 times. So, here we go again with the new key:

```
go run ./cmd/encipher -key \
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1 \
< imgdata.txt >enciphered.bin
cat header.txt enciphered.bin >test.ppm
```

And the result:



Hmm. It's *okay*. Not great.

You can definitely still see at least the outlines of the original image. It's scrambled the colours a bit, but it's nowhere near the ideal "random noise" image that we were going for.

But this was enciphered using the longest practicable key, generated using the randomest numbers in the universe. So why does it still *really* look like a devil gopher?

What's wrong with this picture?

Well, the answer is simple, and we've already talked about it in previous chapters: we're repeating the key. Even though 32 bytes is long for a *key*, it's a tiny fraction of the data in our image: about 0.05%.

This 32-byte repeat is clearly visible in the "enciphered" image as a regular stippling effect. Because there are regularities in the original image with a much longer "wavelength", if you like, this 32-byte perturbation doesn't do much to hide them.

So, while *some* encryption is clearly happening, is this good enough? I think not. If Alice wants to send top-secret blueprints or schematics, for example, it's no good just stippling them a bit and shuffling the colours.

Line drawings and diagrams would still be clearly visible to Eve in the enciphered message. Maybe even text, if it's in a big enough font.

What can we do? Use a longer key, maybe?

Mallory in the middle

Actually, there's nothing wrong with the key. The problem here is the *operating mode* of our cipher. The way we handle each block is pretty simple-minded: we just encipher (or decipher) it independently, by combining it with the key.

ECB: a block-headed operating mode

That means, as we've already seen in previous chapters, that a given *input* block always produces the same *output* block. In fact, you could make enciphering and deciphering very fast by simply generating all the possible input-output mappings in advance, as a lookup table.

Such a *code book* would be easy to use. Given a block of plaintext, for example, Alice could just look it up in the book. Every 32-byte block is equivalent to some large integer, as we know, so all she'd need to do is find this page number in the book, read off the corresponding block of ciphertext, and send it to Bob.

The trouble, as we've just seen, is that the blocks are typically very small relative to the message. That means if there is structure in the message on a larger scale, our scheme can't do much to hide it.

For example, in the devil image, there are long sequences of pixels of the same colour, such as the white background, or the red body of the gopher. Wherever a block of white pixels occur together in the plaintext, they'll produce a certain block of pixels in the ciphertext, and that ciphertext block is always the same.

So, while Eve might not be able to reconstruct the precise original colour of those pixels, just the repeating pixel patterns *themselves* are enough to give

away a lot of valuable information about the image. We need to fix that.

Specifically, we need a better operating mode. It turns out the one we're using, namely "just combine each block with the key", has a name: it's called Electronic Code Book (ECB). That's because it acts like a physical code book, as we've seen, in that the same input always maps to the same output.

This is a serious weakness, because if Eve can collect or generate such a code book, she can use it to trivially decipher any message. All she has to do is look up the number corresponding to a particular ciphertext block, and read off the corresponding plaintext.

Introducing Mallory

And there's another problem with ECB-encrypted messages. To illustrate it, let's introduce another member of the traditional cryptographic theatre company: Mallory.

Like Eve, Mallory is an adversary of Alice and Bob's. Whereas Eve can only read Alice and Bob's ciphertexts, Mallory can read them and also potentially modify them in transit.

Block replay

Suppose Mallory sells Alice a car for \$1000, and Alice pays for it with an electronic funds transfer via her banker, Bob. This is encrypted, naturally, but Mallory can spy on and record the transmitted ciphertext. While they can't recover the plaintext without the key, it doesn't matter, because they already know what it says: in effect, "transfer \$1000 from Alice's account to Mallory's."

Mallory can now carry out a *replay attack*: they simply re-transmit the same ciphertext block to Bob the banker's computer. Bob obediently deciphers the message and carries out what he believes is Alice's new instruction. Jackpot for Mallory! Another \$1000 in their account.

And Mallory can keep doing this as often as they like. All they need to do is record and replay the single block of the ciphertext conversation that contains Alice's transfer instruction.

Because the cipher is operating in ECB mode, the same message always enciphers to the same ciphertext block, so the bank has no way to know that the replayed blocks are not genuine.

Dropping and modifying blocks

Mallory can work other kinds of mischief with Alice's cipher traffic. For example, suppose Mallory buys a laptop from Alice for \$1000, and pays by electronic transfer. If they can identify the specific block containing the transfer instruction, they can simply delete it from the sequence of blocks, so that Bob never receives it.

As far as Alice knows, though, the transfer has been made. She won't realise what has happened until the money fails to show up in her account, by which time Mallory may be long gone.

In this case, the vulnerability is slightly different: it depends on the fact that each block in ECB mode is completely independent of those that went before. Thus, there's no way for Bob to know that a block has been removed by Mallory, just as there's no way for him to know if Mallory is repeating a block.

Mallory can also modify blocks, though how useful this is to them depends on the specific cipher in use, and other details such as the padding scheme. Suffice it to say that ECB mode offers no protection against this general class of attack, which is why it's rarely used in real cryptographic systems.

We need something a little more sophisticated. Specifically, we need to make it so that the same plaintext block *doesn't* always produce the same ciphertext block.

More sophisticated modes

So what other operating modes *could* there be? I mean, we still need to combine each block of plaintext with the key, so what else could we throw into the mix, to spice things up a bit? It would need to be something that changes with every block, to avoid the repetitious “code book” problem.

Counter mode (CTR)

One simple idea is to use a *counter*: just increment some number every time we encipher a new block, and combine that number with the key and plaintext. Just using a simple sequence number on its own wouldn’t add much spice to the dish, admittedly, but with a few extra tweaks, it can be quite effective. This mode of operation is called *CTR* (it doesn’t stand for anything, it’s just short for “counter”).

In ECB mode, the only input to each block is the plaintext and the key. In CTR mode, we add an extra input: the counter value. However, we could invent other modes where the extra input doesn’t have to come from a counter: it could just be random data, or the wall-clock time, or any other value that changes frequently.

Nonces

This extra input is sometimes called a *nonce*, from the expression “for the nonce”, meaning some one-off value that won’t be re-used. The point, of course, is that if we did re-use the same value for every block, we’d just be back in plain old ECB mode, and we wouldn’t be able to foil Mallory’s replay attacks.

The nonce doesn’t need to be secret. Bob needs to know it for each block, so that he can decipher the block. If it’s a counter value, he can just keep count of the blocks, or we could include the nonce value before the beginning of each block.

Either way, Mallory will know it too, but that doesn’t matter: without the key, they still won’t be able to decipher any block. Bob, though, has the key *and* the nonces, so he can.

Cipher Block Chain (CBC)

One interesting value we could use as the extra input to each block is... the previous block! This is rather elegant, because we don't need to maintain a counter, or otherwise generate a unique nonce value for each block. Instead, we simply use the ciphertext of the previous block that we just enciphered.

In this mode, then, the input to the cipher for each block consists of:

1. The key
2. The plaintext for this block
3. The ciphertext for the previous block

These are all combined to produce the ciphertext for *this* block—which will be used as the nonce for the next block, and so on.

This cleverly avoids the “same plaintext, same ciphertext block” problem. Each new block of ciphertext we generate will be different, even if the plaintext is the same as the previous block’s. And if Mallory tries to replay a block, Bob won’t be fooled. He’ll know that something nefarious has happened, because he won’t be able to decipher the repeated block or anything after it.

Mallory also can’t drop, add, or modify blocks, for the same reason—at least, not without being detected by Bob. Each valid block depends for its decryption on the previous one, and so on, forming a chain of linked blocks. A *blockchain*, indeed, which is where that term comes from.

Thus, this operating mode is known as Cipher Block Chain (CBC).

Initialization Vector (IV)

However, this only partially solves the “same plaintext, same ciphertext” problem:

The block-by-block problem is solved by making each block’s encrypted value depend on all of the previous blocks but the larger message-by-message problem remains because the same full message

will still encrypt to the same value.

—Azeem Bande-Ali, “[Intuition for Cryptography](#)”

In other words, if the entire message is the same as one Alice has previously sent, then the first block will be the same, so its ciphertext will be the same, too. And since every block depends on the previous one, all the blocks will end up the same. Mallory could simply replay the entire message instead of an individual block.

That’s because, for the first block, we *have* no previous ciphertext to use as the nonce input. What could we do? Where could we get some suitable data from?

The answer is that it doesn’t matter: random data will do. We just need a block’s worth of junk bits to get the blockchain started. Again, it doesn’t have to be kept secret: we’ll be sending this *initialization vector* (IV) to Bob in the clear, along with the ciphertext message.

Generating a secure IV

To generate the IV, we can just grab some bits from the system entropy pool, or toss some coins. Now the same message *won’t* produce the same ciphertext even with the same key, so long as it has a unique IV.

If you’re *really* paying close attention, though, you might wonder what happens if Mallory modifies the IV in transit. Wouldn’t that make the whole message unreadable for Bob?

Well, no: remember, the IV is only used to encipher the *first* block. After that, the input to each block is the ciphertext of the previous block, which won’t have changed. Yes, Bob will fail to decipher the first block, or decipher it incorrectly, but he’ll still be able to read all the other blocks.

And there are ways that Alice and Bob can protect their messages from being secretly modified by Mallory *at all*, which—spoilers!—we’ll talk about later on in this book.

Implementing CBC mode

First, let's see how to improve our shift cipher by operating it in CBC mode. We *could* write a bunch of code to do everything we've just described: mingle each block's plaintext with the previous block's ciphertext, and so on. But there's no need: once again, the standard library has done the work for us.

The `crypto/cipher` package contains code that will take a block cipher—that is, anything that implements `cipher.Block`—and give us an “encrypter” object that operates the cipher in CBC mode. That object, naturally enough, is an implementation of `cipher.Mode`.

Just as a quick reminder, since we last saw it a few chapters ago, here's an edited snippet from the code we currently have in our `encipher` tool:

```
block, err := shift.NewCipher(key)
...
enc := shift.NewEncrypter(block)
...
enc.CryptBlocks(ciphertext, plaintext)
...
```

Taking this step by step, here's what happens:

1. We use the key to create a cipher
2. We use the cipher to create an encrypter
3. We use the encrypter to encipher the plaintext

Enciphering in CBC mode

We actually don't need to change the code that much to use CBC instead of ECB mode, now that we've discovered ECB is a bust. The main difference

is that, instead of calling our own function `shift.NewEncrypter`, we'll call the standard library's `cipher.NewCBCEncrypter`.

However, we now need to supply an IV, along with our block cipher, so the call will look something like this:

```
enc := cipher.NewCBCEncrypter(block, iv)
```

We can use the `crypto/rand` package to generate that random `iv` value. Over to you for the next part, then!

GOAL: Update the `encipher` tool to use CBC mode instead of ECB.

HINT: As usual, let's start by updating the tests. We won't need `shift.NewEncrypter` or `shift.NewDecrypter` any more, so let's remove those tests, plus the code they test. That cleans up the `shift` package nicely.

At present, I don't think we need to write any new tests, so let's go straight on to the `main.go` file for the `encipher` tool and see what we can do there.

We already know that we'll be calling `NewCBCEncrypter`, and as a result we'll need some random data to act as the IV for the first block. We can get that from `crypto/rand`, using the `Read` method:

```
iv := make([]byte, shift.BlockSize)
_, err = rand.Read(iv)
// handle error
```

Once we've successfully created our CBC encrypter, we can call its `cryptBlocks` method, as before, to do the actual encryption.

There's just one problem left to solve: we need to output not just the ciphertext, but the block of IV data too. Bob will need this to decipher the message.

It doesn't matter how we get this IV block to him, but one obvious way is simply to make it the first block of the message. So the program should output the IV first, followed by the ciphertext data.

We'll handle this in the decipher program by assuming that the first block we read will be the IV, and the rest will be the ciphertext.

The updated encipher program

SOLUTION: Let's start with the preamble, which is exactly the same as for the previous version of encipher:

```
func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := shift.NewCipher(key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

([Listing shift/8](#))

So far, so good. Now, there's no point generating an IV if it turns out we weren't able to read all the plaintext, so let's do that first. Again, that's exactly the same as before:

```
plaintext, err := io.ReadAll(os.Stdin)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
```

([Listing shift/8](#))

Now we can read a block's worth of random data to create our IV, and write it out:

```
iv := make([]byte, shift.BlockSize)
_, err = rand.Read(iv)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
os.Stdout.Write(iv)
```

([Listing shift/8](#))

Finally, we'll create the encrypter, pad the plaintext to the required length, encrypt it, and write out the ciphertext:

```
enc := cipher.NewCBCEncrypter(block, iv)
plaintext = shift.Pad(plaintext, shift.BlockSize)
```

```
ciphertext := make([]byte, len(plaintext))
enc.CryptBlocks(ciphertext, plaintext)
os.Stdout.Write(ciphertext)
```

([Listing shift/8](#))

Not too hard in principle; there are just a lot of moving parts to keep in mind. Let's try the program out and see what it does to our tiger text:

```
go run ./cmd/encipher -key \
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1
\
< tiger.txt > enciphered.bin
```

Previously, when we enciphered this file in ECB mode, without any IV preamble, it was padded to 480 bytes. Now we should have an extra block's worth of data, making 512 bytes in all. Let's see:

```
wc -c enciphered.bin
```

```
512 enciphered.bin
```

Well, the numbers check out, but the real test will be if we can *decipher* this file. Over to you again!

GOAL: Update the `decipher` tool to use CBC mode.

Updating the `decipher` program

HINT: This should be a lot easier now that we've figured out how enciphering with CBC works. Again, everything's the same up until the

point after we've read the input (in this case, it's the ciphertext).

We now need to extract the IV, which we'll assume is the first block of the data we've just read. Once we have that, we can create the CBC decrypter.

We'll then call its `CryptBlocks` method to decrypt the remainder of the ciphertext (skipping the first block, which is the IV). Finally, we'll unpad the resulting plaintext and write it out.

Can you see what to do?

SOLUTION: Again, let's start with the preamble of getting the key, creating the cipher, and reading the ciphertext data:

```
func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := shift.NewCipher(key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    ciphertext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
```

```
    os.Exit(1)
}
```

([Listing shift/8](#))

Now we need to extract the first block's worth of data into the `iv` variable:

```
iv := ciphertext[:shift.BlockSize]
```

([Listing shift/8](#))

We need a `plaintext` slice that's just one block's worth shorter than the `ciphertext`, since we're discarding the IV:

```
plaintext := make([]byte, len(ciphertext)-shift.BlockSize)
```

([Listing shift/8](#))

Great. Now we can create the decrypter, decrypt the remaining `ciphertext`, unpad the result, and write it out:

```
dec := cipher.NewCBCDecrypter(block, iv)
dec.CryptBlocks(plaintext, ciphertext[shift.BlockSize:])
plaintext = shift.Unpad(plaintext, shift.BlockSize)
os.Stdout.Write(plaintext)
```

([Listing shift/8](#))

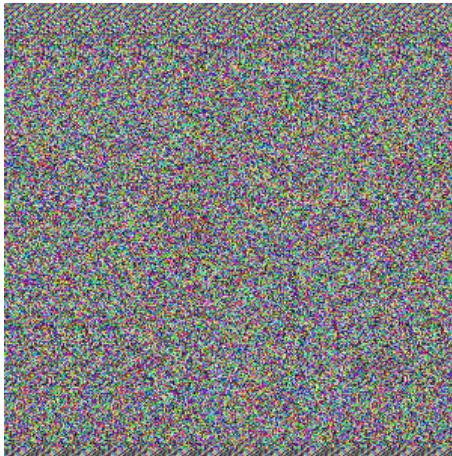
This looks promising, but there's only one way to really know if it works: try to decipher the message we just encrypted.

```
go run ./cmd/decipher -key \
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1 \
< enciphered.bin
```

The tiger appears at its own pleasure...

The devil in disguise

It's nice to see that after all that scrambling and unscrambling, we can still get an intelligible message out at the other end. So let's see how switching to CBC mode affects the enciphered version of our devil image:



Wow! That's really made a difference. This now looks far more like the “random noise” image than the original devil gopher. And that's not surprising, because as we know, the ciphertext for each block is as much affected by the ciphertext for the *previous* block as it is by the plaintext.

As a result, every block looks different, even when there are strong regularities in the original image. While our cipher is by no means perfect, now we're at least operating it in a reasonably secure mode that's much better than the simplistic ECB.

15. Hashing

For the first time he perceived that if you want to keep a secret you must also hide it from yourself.

—George Orwell, “[Nineteen Eighty-Four](#)”



In the previous chapter we talked about the kind of problems that Mallory could cause by intercepting and modifying Alice and Bob’s cipher traffic: for example, deleting certain blocks, or repeating or inserting new blocks.

Message integrity

Adopting a more sophisticated operating mode such as CBC addresses these problems to *some* extent, by linking all the blocks together in a chain. It still doesn’t prevent Mallory messing around with the blocks, but it at least makes it harder for them to avoid detection.

The night is dark, and full of errors

For example, if Mallory alters one bit in the ciphertext of a given block, Bob will find that his decryption program is unable to decipher that block, or any that come after it. That's as it should be, if the cipher is a good one and being operated securely.

But in real life, communication channels are noisy and imperfect: bits could be dropped or flipped due to thermal noise, cosmic rays, or loose cables.

TCP/IP, the internet communication protocol, includes error-correcting machinery for just this reason. It can detect when a packet (the equivalent of a block) has been corrupted, and ask for it to be retransmitted as many times as necessary, until it makes it over the link intact.

We don't have anything like that in our cipher scheme. Yes, if Alice sends her ciphertext data over TCP/IP, it *should* reach Bob without too many egregious bit-level errors. All the same, we have to assume some non-zero error rate, even in the absence of a malicious Mallory.

Bit-flipping and block-dropping

Once Mallory enters the picture, the problem becomes even worse. By flipping bits and making Alice's messages partially unreadable, they can cause a lot of trouble. Bob may have to ask Alice to retransmit her message several times, and this in itself can weaken the security of the conversation. At best, it causes annoying delays and eats up valuable bandwidth.

At worst, Mallory may be able to interfere with the message in undetectable ways. Some cipher schemes are vulnerable to attacks where flipping just the right bits, in just the right way, causes Bob to decipher an incorrect plaintext. Depending on the message, and the situation, that might be harmless. Then again, it might not.

An even simpler attack against a CBC-encrypted conversation is for Mallory to drop one or more blocks from the end of the message. Bob has no way to detect this, because there are no blocks following the deleted ones, so he won't be tipped off by failing to decipher them.

He *might* be suspicious that the message ends abruptly, or even in the middle of a sentence. Alice could help him out by always ending her messages with some agreed word or phrase. For example, in the early days of telegraphy, non-words such as “ENDIT” were sometimes used for this purpose, since they wouldn’t occur as a normal part of the message.

Cryptographically, though, this is a bad idea, as we already know. Any stereotyped or predictable text gives Eve a crib, especially if it occurs at the beginning or end of the plaintext. We need to do a bit better than a simple end-of-message indicator.

Integrity checking

Let’s frame the problem this way. How can Bob know that the ciphertext message he received is exactly the one Alice sent?

The obvious ideas don’t work. For example, suppose she begins each message with a number giving the length of the whole message in characters. That won’t stop Mallory shortening or lengthening the message.

If the length prefix is in clear, then Mallory can just change it to match the altered message. If it’s in cipher, then they can use it as a crib, because Alice can’t easily conceal the length of the message.

In any case, Mallory could simply add junk data to make up the required length. We need some scheme where the “tamper-proof” metadata isn’t easily deduced or altered by Mallory, and it’s connected in some robust way with the actual *content* of the message.

In other words, Alice should be able to do some computation on the message, and send Bob the result. He should then be able to do the same computation and compare his result with Alice’s. If they differ, then there’s a very high probability that the message has been corrupted.

Digests and hashing

The computed value is called a *digest* of the message, and as the name suggests, it's usually shorter than the message itself. In fact, a digest usually has some fixed length—for example, 256 bits—regardless of the size of the message it represents.

The problem of taking some arbitrarily long string of bits (a message), and turning it into some representative fixed-length string is called *hashing*, as in “corned-beef hash”, or “making a hash of things”.

As the name suggests, it involves mixing or muddling up the bits in some way. So, how do we make a hash of a string of bits?

Well, there are many possibilities: many imaginable hash *algorithms*. And that makes sense, since there are many different applications for hashing, and the choice of algorithm depends on what you want to use it for.

Hash tables

For example, one common application is the *hash table*: essentially, an index for quick lookups. Suppose you have a million files stored on a particular disk. How can you find the particular one you’re looking for?

One simple idea is to just start at the first byte of the disk and keep on reading sequentially until you find the start of the file you want. That’s easy to implement, but it performs *really* badly. On average you’d have to read half of the entire disk per file access, and that sucks.

Of course you wouldn’t do it this way. Instead, you’d keep a list of all the files by name, and link each one to its starting position on the disk.

This lookup table (or *directory*, which is where that term comes from) means that you can jump straight to the beginning of the file, without reading any of the intervening data.

This is great, but what happens when the directory itself becomes large? In our million-file example, a simple-minded *table scan* means we have to read every directory entry until we find the one associated with the target

file. On average, that means reading half a million entries per file access. Not as slow as reading half of the whole disk, but still not great.

What can we do?

Buckets and distribution

Ideally, we'd break up the directory into a number of smaller chunks—known as *buckets*—and scan only the chunk that contains the entry for the target file. But how do we know which chunk that is?

We could imagine allocating a chunk for each initial letter of the filename, for example, from A to Z. Then, given a target filename to find, all we need to do is look at its first letter, go straight to the relevant chunk, and scan it until we find the file. That's a hash algorithm, of a kind.

In practice, this wouldn't work very well: the buckets wouldn't be equally populated. The "T", "A", and "O" buckets would be crammed to overflowing, since these are the most common initial letters of English words. Meanwhile, the "X" and "Z" buckets would be mostly wasted space. That's why physical filing cabinets don't allocate equal space to each initial letter: instead, they often have a single drawer for "VWXYZ", and so on.

In other words, our imaginary hash function—taking the first character of the filename—doesn't produce a *uniform distribution*. Instead, it suffers from excessive *clustering*.

This doesn't make it unusable, but we can do better. The more uniform a distribution we can produce, the more efficient our hash table will be.

Collisions

You could imagine being really mean to the owner of such a disk addressing system by supplying them with a bunch of files whose names are all specially crafted to produce *collisions*: that is, to make them all end up in the same bucket.

For example, such a *pathological key set* might contain files whose names all begin with “Z”. Then almost all of the hash table would be empty, while the “Z” bucket would be horribly overloaded. Thus, every file access would effectively require a full table scan: just what we wanted to avoid.

So when we’re thinking about the performance of a given hash function, we have to consider not just the average or common case, but the potential worst case—especially if we have an adversary who wants to make life difficult for us.

For hash tables, *some* collisions are okay: indeed, they’re useful. If there were no collisions at all, every file would hash to a separate bucket, and we wouldn’t gain any advantage from having the hash table. On the other hand, if the buckets grow too full, then searching them for the key we want takes a long time. So we want a moderate rate of collisions for typical key sets.

Cryptographic hashing

That’s certainly not true in cryptographic applications, though. The whole point of a message digest is that every message should produce a different hash. Thus, we want a very large hashspace.

Some collisions are still inevitable, of course: there are only so many possible hash values for a given number of bits, just as there are only so many possible keys.

Even an ideal hash algorithm, then, can’t do better than a certain minimum collision rate. It’s easy to work out what that is: if we start with a given message, the probability of randomly generating another message with the same N -bit hash is 1 in 2^N .

Alternatively, if we pick *both* messages at random, the chance of them having the same hash is 1 in $2^{N/2}$.

Therefore, we should make N large enough that this rate of collision is acceptable, and use an algorithm that doesn’t increase it too much above the theoretical minimum.

Simple hash functions

Let's put this into context by trying to implement a very simple hash function ourselves. First, what kind of input should it take, and what should it return?

We've been working with arbitrary bytes throughout this book, so let's stick to that. Our hash function will take a []byte of any size, and it'll return a []byte of some fixed size (let's say 8, giving us a 64-bit digest).

A naïve algorithm

What about the algorithm? Well, let's start with something silly: we'll just take the length of the input! We know intuitively that this won't be any good, but let's prove it.

Here's a test:

```
func TestLenHashReturnsExpectedResult(t *testing.T) {
    t.Parallel()
    input := []byte("I love you, Bob")
    want := []byte{
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x0f,
    }
    got := hash.LenHash(input)
    if !bytes.Equal(want, got) {
        t.Errorf("%s: want %x, got %x", input, want, got)
    }
}
```

([Listing hash/1](#))

In other words, hashing the input “I love you, Bob” should produce the value 15, written as a (big-endian) 8-byte slice. Over to you now to make this test pass!

Implementing LenHash

GOAL: Make the test pass by implementing LenHash.

HINT: Computing the length of the input isn’t too hard—we can use the built-in `len` function for that. Actually, the only tricky bit might be turning its result (an `int`) into a slice of 8 bytes.

Remember when we used the standard library `encoding/hex` package to turn a hex value into a `[]byte`? We can use its brother `encoding/binary` to do something similar here.

For example, if we have some slice `digest`, we can encode a value `val` to it like this:

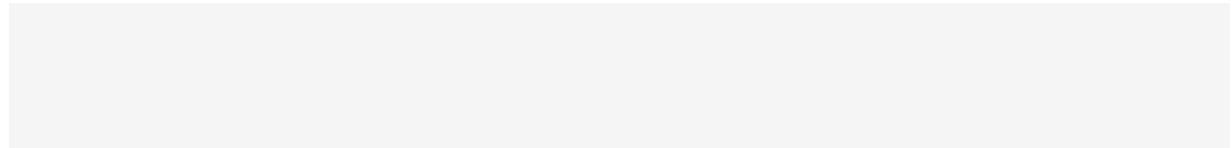
```
binary.BigEndian.PutUint64(digest, val)
```

This will turn the integer `val` into its representative bit pattern, and write those bits into the `digest` slice. For example, if `val` is 15, the contents of `digest` should now be:

`0x000000000000000F`

Have you figured out what to do?

SOLUTION: Here’s something that will do the job:



```
func LenHash(input []byte) []byte {
    digest := make([]byte, 8)
    binary.BigEndian.PutUint64(digest, uint64(len(input)))
    return digest
}
```

([Listing hash/1](#))

Let's add a simple `main` function so that we can try it out:

```
func main() {
    data, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Printf("%x\n", hash.LenHash(data))
}
```

([Listing hash/1](#))

We read the input from `os.Stdin`, as we did with our enciphering and deciphering tools. Then we pass it to `LenHash` and print the result. Here we go:

```
echo "I love you, Bob" | go run ./cmd/hash
```

```
0000000000000010
```

Looks reasonable. The input is 16 characters long, counting the final newline, and 16 in decimal is 10 in hex. So, in theory, Alice could use this tool to compute a hash digest for her ciphertext message to Bob.

Problems with LenHash

But *should* she? Definitely not, and we already know why: collisions are extremely likely. Lots of different messages will produce exactly the same hash. That weakens our confidence that the plaintext hasn't been corrupted, accidentally or on purpose.

It also means it's relatively easy for Mallory to carry out a brute-force attack on the hash. They can just hash lots of random messages until they find one that hashes to the same value as Alice's. Then they can replace Alice's ciphertext with their own message, and Bob will be none the wiser.

And, even though we're using a 64-bit hash ($N = 64$), our hashes have far from a uniform distribution.

LenHash mostly produces pretty low numbers, even for big inputs. In other words, we're not evenly covering the space of possible hashes. Most of the bits in the hash we just computed are zero, and most of the bits for *every* message hash will be zero.

If that's not clear, think about how long a message would have to be in order for all its hash bits to be 1 with the LenHash function. Since there are 64 bits, the message would have to be 2^{64} characters long, or about 17 trillion gigabytes. Alice likes to chit-chat, but even she's unlikely to send messages this long.

That makes Mallory's brute-force search far easier than it should be for a 64-bit hash. Indeed, if we assume that most messages will be smaller than one gigabyte, Mallory only has to search the space of 30-bit hashes, which is billions of times smaller.

Preimage attacks

Even worse, if Mallory knows the hash algorithm, then they don't even need to do a brute-force search. It's very easy for them to deliberately construct a message that produces the same hash.

Such a message is known as a *preimage*, and we want it to be hard to find. Instead, it's trivial. Any message of the same length will do:

```
echo "I hate you, Bob" | go run ./cmd/hash
```

```
0000000000000010
```

Oh no! Now Mallory can substitute a message that says the exact opposite of what Alice meant to send (ignoring encryption for the moment), and Bob will compute the same hash value, so he'll think the message is authentic. This could spell disaster for Alice and Bob's blossoming relationship.

We don't want to make it quite this easy for Mallory to sunder these two hearts, so what would a more sophisticated hash algorithm look like?

A slight improvement

One problem with LenHash, as we've seen, is that the hash only depends on the length of the message, which is something Mallory can easily find out.

What if we made it depend on something about the actual *content* of the message instead? For example, what if we added together the values of all the bytes?

Implementing SumHash

GOAL: Write a function `SumHash` that produces a digest based on the sum of the input bytes. It should pass the following test:

```
func TestSumHashReturnsExpectedResult(t *testing.T) {
    t.Parallel()
    input := []byte("I love you, Bob")
    want := []byte{
```

```
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x04, 0xfb,
}
got := hash.SumHash(input)
if !bytes.Equal(want, got) {
    t.Errorf("%s: want %x, got %x", input, want, got)
}
}
```

([Listing hash/2](#))

HINT: We already have code to take some integer and encode it as a slice of 8 bytes, so all we need to do is loop over the input bytes, adding them up until we get the grand total. Can you see what to do?

SOLUTION: How about this?

```
func SumHash(input []byte) []byte {
    digest := make([]byte, 8)
    var sum uint64
    for _, b := range input {
        sum += uint64(b)
    }
    binary.BigEndian.PutUint64(digest, sum)
    return digest
}
```

([Listing hash/2](#))

Testing SumHash

Let's try this version with our two different input messages. Will they produce different hashes?

```
echo "I love you, Bob" | go run ./cmd/hash
```

```
0000000000000505
```

```
echo "I hate you, Bob" | go run ./cmd/hash
```

```
0000000000004f1
```

This looks promising! Our new hash algorithm doesn't produce the same output if the inputs are the same length. So that makes it harder for Mallory to construct a fake message that hashes to some given value. But is it still possible? Let's try.

A preimage attack on SumHash

Suppose we want to produce the hash 0505, which is 1285 in decimal. It doesn't matter what our input bytes *are*, or even how many of them there are, so long as they sum to 1285. Let's start with a bunch of z characters:

```
zzzzzzzzzz
```

That's 10 zs, and the character code for z is 122, so $10 \times 122 = 1220$. That gets us close. We're just 65 short, and that's the code for A, so let's try this input:

```
echo "zzzzzzzzzzA" | go run ./cmd/hash
```

```
000000000000050f
```

Darn. We forgot about the newline character that echo silently appends to its argument.

Well, the code for newline is 10 in decimal, so we need the character that's 10 less than A, which is 7. Here we go:

```
echo "zzzzzzzz7" | go run ./cmd/hash
```

```
0000000000000505
```

Bingo! We've broken the *preimage resistance* of our hash function. Sure, Bob might be a little confused to receive a message like this, but when he checks the hash value, it'll be the same as the one Alice computed. We still need to do better.

The avalanche effect

Just like `LenHash`, the distribution of hashes produced by `SumHash` isn't very good. The sum of even quite long messages will still be a fairly low number, so our hashes will end up all clustered together in one tiny corner of the hashspace, like shy teenagers at a school dance.

Another problem is that small changes in the message only make small changes in the hash. For example, if we replace an "A" in the message with a "B", the hash value will only change by one.

That's not just a statistical problem: yet again, it makes it easier for Mallory to construct a preimage with the desired hash. All they need to do is get fairly close, and then make smaller and smaller tweaks to the input until they have the exact hash they want.

By contrast, a good cryptographic hash function would exhibit what's called the *avalanche effect*. Tiny changes in the input should cause large changes in the output, like a snowball rolling down a mountain and eventually building up into an avalanche. `SumHash` certainly doesn't have this property.

Real hash algorithms

We could imagine other simple-minded hash algorithms that are a *little* better than the ones we've seen so far. For example, `ProductHash` would distribute the hashes slightly more widely than `SumHash`, and small changes in the message would perturb the hash more.

But there's clearly room to do a *lot* better.

MD5

One much better hash algorithm is Message Digest Method 5 (*MD5*), designed by cryptographer Ron Rivest (he's the "R" in "RSA", which we'll come to later).

In comparison with something like `SumHash`, MD5 is astoundingly sophisticated. Even just to describe it in pseudocode takes something like [90 lines](#).

MD5 breaks the input message into blocks, and uses each block in turn to modify the final hash value, by putting it through a tortuous sequence of bit-twiddling operations. These are so complicated that for about five years, MD5 was considered state-of-the-art and used widely in some very high-stakes cryptography.

Time and technology moves on, though, and nowadays you can generate MD5 collisions with a simple brute-force search in one or two seconds on an ordinary desktop computer. That makes MD5 completely unsuitable for any cryptographic purpose.

Thank u, next?

SHA-1

A much stronger replacement for MD5 was Secure Hash Algorithm 1 (*SHA-1*), which is even more complicated. And, again, for a while SHA-1 was considered secure enough for cryptographic work.

Pretty soon, though, researchers began to find more and more effective attacks against SHA-1. A study in 2015 found that its strength had been reduced to the point where only about \$100,000 of cloud compute time was needed to generate a collision. That sounds a lot, but it's just pocket change for a sufficiently motivated Mallory.

So, while SHA-1 is considerably better than MD5, it's still not good enough for cryptography *today*. Next please.

SHA-256

Secure Hash Algorithm 2 (*SHA-2*) is an extension and improvement of SHA-1, and there are variants for different hash lengths, of which SHA-256 is the most popular.

This makes sense, since 256 bits is, as we've discussed, already long enough to make brute-force attacks impractical. Fewer bits would be insecure, but more would be unnecessary.

There are currently no known practical attacks on SHA-256 hashes. That is not, of course, to say that none will be found: in fact, I guarantee that sooner or later one will. But it's secure *today*, and that's all we can reasonably hope for from any algorithm.

If you're really worried, you can use *SHA-3*, which is newer and fundamentally different from the SHA-1 and SHA-2 families. But SHA-256 is pretty much the de facto standard.

Go provides an implementation of SHA-256 in the `crypto/sha256` package, and it's nice and easy to use. All you need to do to hash some data is:

```
hash := sha256.Sum256(data)
```

Password hashing

Another common use for hashing is to provide a *zero-knowledge proof*. In other words, to able to prove that we know a secret, but without revealing the secret itself.

Zero-knowledge secret storage

Think about storing passwords, for example. Suppose we're designing a system, such as a bank, where the user—Alice—logs in by entering her password. How does the system know that this is the same password she originally set when she created her account?

The obvious idea—just storing the password somewhere—would be a fatal mistake. If we stored passwords in clear, then the security of the whole system would depend on the security of that password list.

Ideally, we'd put it on a USB key and bury it at a lonely crossroads at midnight, or something, and never reveal its location. But clearly we can't do that: we need to consult the password list every time Alice logs in!

There's no point protecting secret information with a password which itself then has to be kept secret. That's just kicking the can down the road. Worse, if Alice accesses the system over the network, then she has to send her *password* over the network, where Eve can easily intercept and steal it. Bad times.

So, here's the problem. How can Alice prove to the bank—Bob—that she knows the secret password, without sending Bob the password, and ideally without Bob knowing the password either?

At first glance, it seems impossible.

Password hashing requirements

That's where hashing comes in. Suppose when Alice signs up with Bob's bank, the system stores, not her password, but some *hash* of her password.

Then, some time later, when she comes to log in, the bank's website can prompt her for the password, compute its hash, and compare it against the

original stored hash.

This is an elegant solution, because Alice’s password never has to travel over the unsecured network: not when she registers her account, and not when she logs in. The hash can be computed and sent by code running in Alice’s web browser. The bank never knows or stores her actual password, only its hash.

So what hash algorithm should we use? Clearly we don’t want one with a high collision rate. If Eve can choose a password whose hash collides with Alice’s, then she could log in as Alice.

In fact, if Eve can gain access to the list of stored hashes, she can brute-force a preimage for *any* user’s password. We need to make that as difficult as possible.

Rainbow tables

Of course, we don’t want the password hash list to fall into the hands of Eve, or other evil-doers, but we have to assume that’s going to happen, and keep our users safe even if it does. So how would Eve’s brute-force attack work, exactly?

One obvious thing she could do is just try hashing every English word in the dictionary and see if it matches one of the hashes in the password list. She only needs to generate this set of hashes once, and then she can use it as a lookup table for any password list that she comes across.

Needless to say, this doesn’t sound like a good situation. In fact, it sounds familiar, doesn’t it? It sounds just like the problem we identified with block ciphers in ECB mode: that the same block always encrypts to the same ciphertext.

In this case, the same password always hashes to the same digest, but Eve’s exploit is identical. She just creates a “code book” of hashed passwords: all the words in the dictionary, plus the usual bad passwords like pa55w0rd and friends.

With this code book—colourfully known as a *rainbow table*—looking up a stored hash and identifying the password that generated it is trivial.

Salting

We can solve this problem in the same way that we dealt with ECB. Instead of just hashing the password on its own, we mix in some random, unique data with it, and *then* hash it.

The result is a different hash value for every password. Indeed, even if two users happen to have the same password by chance, their stored hash values will be different.

As with a block cipher, it doesn't actually matter how the additional data—the *salt*—is generated. We have to include it with the stored hash anyway, or the banking system wouldn't be able to check Alice's password when she logs in.

“But what’s the point of that?” you might well ask. “If we’re giving Eve the salt along with the hash, couldn’t she still carry out her dictionary attack by hashing each word in conjunction with the salt?”

Yes, she could. But her job is made very much harder, because now she has to re-hash every word in the dictionary *for every password* that she wants to crack. Because the stored salt is different with every stored hash, her one-size-fits-all rainbow table is rendered useless.

It’s certainly not impossible for her to do this, of course, but even with a fast hash function, hashing is very much slower than just *comparing* pre-computed values in a rainbow table.

Slow down, you move too fast

And that’s one final thing to note about password hashing that makes it special. In almost all other applications of hashing, we want our hash function to be as fast as possible. That is to say, we want it to run very

efficiently on general-purpose hardware, or to be easily accelerable by special-purpose hardware, such as GPUs.

With passwords, on the other hand, we want the hashing process to be as slow as possible! That sounds strange, but remember, we only need to *legitimately* hash the password once a day or so, when Alice logs in.

Alice won't notice or mind a delay of a few milliseconds in checking her password. Only bad actors such as Eve need to hash passwords very quickly, in vast numbers.

So we want a hash function that is relatively slow, inefficient, expensive, or a combination of all three. SHA-256 *wouldn't* be a good choice in this case, because it's extremely fast and efficient, and it's easy to obtain special-purpose hardware for computing it even faster. Indeed, there are strong financial incentives for this, as we'll see in the next chapter.

Instead, there are dedicated password hashing algorithms such as [bcrypt](#) and [scrypt](#) that make much better choices for storing hashed passwords securely. We needn't say much more about them here, but it's good to know they exist.

16. Coins

*I speak the pass-word primeval, I give the sign of democracy
By God! I will accept nothing which all cannot have their counterpart
of on the same terms.*

—Walt Whitman, [“Song of Myself”](#)



Another important application of hashing is in digital ledgers, or *blockchains*, such as Bitcoin. As we saw in a previous chapter, a cipher operating in CBC mode creates a blockchain, in a sense. The term “blockchain”, though, is usually used in the context of a *cryptocurrency*.

Cryptocurrency

Suppose Bob wants to branch out beyond traditional banking (Alice was his only customer anyway), and create his own secure digital currency: “Bobcoin”, perhaps. How could he do that?

Let there be “Bobcoin”

The simplest way would just be to declare to the world that Bobcoin is now a thing, and invite members of the public to exchange their dollars or pounds for the equivalent amount of Bobcoin.

There's nothing to stop anyone creating their own *fiat* currency (Latin for “make it so”) in this way, and of course that's how all currencies got started.

Traditionally, the exchange is done by a central bank—in this case, the Bank of Bob—which determines the rate of conversion. For example, Bob might declare that one Bobcoin is convertible to one US dollar. Alice pays in \$100 and Bob credits her with a balance of 100 Bobcoin.

So far, so good, but what can Alice do with this balance?

Well, she can buy a pizza (from Pat, let's say). No physical tokens, such as banknotes, need to change hands. Alice simply uses Bob's banking system to transfer, as it might be, ten Bobcoin from her account to Pat's. Pat checks her own balance, sees that the ten Bobcoin has been added to it, and she sends the pizza right along to Alice's house.

Then maybe Pat uses the Bobcoin she just earned to buy a really cute torque wrench she's had her eye on. And so forth. Round and round the Bobcoin goes: where it stops, nobody knows.

Or rather, Bob knows, because he keeps careful track of everyone's balance. If Pat decides that she likes the reassuring feel of crisp, green paper (who doesn't?) she simply asks Bob to convert her ten Bobcoin into \$10, and then withdraws it from a nearby ATM.

The problem of trust

So far, so good, but we don't really need a blockchain for this. Bob can just maintain a record of all the transactions and balances in the usual way.

But, if you think about it, it's difficult for Bobcoin to scale up. If it really took off and everyone in the world started using Bobcoin for every online

transaction, that would place a huge load on Bob's servers.

Worse, it makes Bob a single point of failure for the global financial system. If Bob's electricity goes out, commerce everywhere is brought to a shuddering halt. That would be bad, but even if Bob installs a UPS, which he should, there's another problem. Why should everyone *trust* Bob?

They shouldn't. Bob's a good guy, I'm sure, but who's to say that the prospect of stealing literally all the Bobcoin in the world wouldn't turn even Bob's head? He could be threatened or blackmailed into handing over the keys to the digital vault, or he might just lose them by accident.

Even if Bob is incorruptible and careful, he can't know for sure that someone hasn't sneaked into his system and left themselves a little back door, and nor can anyone else. As long as all the transactions are kept private, no one can see if anything nefarious is going on.

So the slightly surprising solution is for Bob to make them all public. He can take the digital *ledger*—the ongoing record of all Bobcoin transactions—and put it on a public file server for all to see.

Now if Bob, or anyone else, steals money, he can't conceal those transactions: they're a matter of public record, literally.

A distributed digital ledger

That removes the Bank of Bob as a single point of failure, but now we have a new one: the public ledger server. No problem: we'll distribute copies of the ledger all over the place—perhaps to every computer in the world that uses Bobcoin for transactions.

Cool. But what happens when a new transaction needs to be recorded? Won't the ledgers now be out of sync?

Yes, but only temporarily. Anyone making a transaction can send out a message about it on the Bobcoin network. All nodes on the network can listen for messages about new transactions, and use them to update their own local copy of the ledger.

This *distributed* ledger will be, in the term of art, *eventually consistent*. That is, if you wait long enough, every transaction will show up in every copy of the ledger, and everyone's balances will agree. Not moment-by-moment, for sure, but within a few seconds or minutes, which is fine.

Security

Or is it? I mean, electronic transactions can happen very fast. What's to prevent some scammer, Sam, from spending the same Bobcoin twice in succession?

For example, suppose that Sam has a starting balance of ten Bobcoin, and they use it to buy a pizza from Pat. But then, moving quickly, before the transaction has had a chance to propagate across the network, Sam spends the same ten Bobcoin on an ice cream from Irene.

The double-spending problem

The distributed ledger will eventually reconcile these two transactions, of course. At that point, Sam will be *overdrawn*—they'll have a negative balance—but maybe they don't care about that. After all, they *have* the pizza and ice cream. It's Irene who's actually out of pocket—or Pat, depending on which transaction the network accepts first.

If Sam wants to make further transactions in the future, there's nothing to stop them just opening a new account with a fake name. Then they can overdraw *that* account, and so on.

This would be bad news for Pat and Irene, and junk food vendors everywhere, not to mention the credibility of the Bobcoin system.

Clearly we need some way for Irene to verify that Sam actually has the money to pay for their ice cream. How could we do that?

Ordering transactions

Suppose Sam opens their account with a deposit of ten Bobcoin, and then makes two transactions: spending ten Bobcoin on a pizza, and spending ten Bobcoin on ice cream.

The second of those two transactions should fail, because Sam doesn't have any money. But, given that Irene's computer receives these transactions after a delay, and in an unknown order, how can she determine which one came first?

Let's assume that the stream of transaction data is divided into equal-sized blocks, for convenience. Now the problem reduces to ordering those blocks. To put it another way, we need a way to *chain* the blocks so that we can detect any out-of-sequence (or modified) block.

Does that sound familiar? It should.

We already know about a kind of blockchain whose ordering is fixed, because each block depends on the previous one. That would be a block cipher operated in CBC mode.

But we're not encrypting the Bobcoin transactions, because they must be public. How could we use the contents of a given Bobcoin block to produce a unique input to the next block, in a way that makes it impossible (or very difficult) to synthesise fake blocks?

That sounds a lot like a job for a hash function, doesn't it?

The blockchain

And that's what we use. When each *node*—each computer in the Bobcoin network—has received a block's worth of transactions, it starts trying to compute the next block. It hashes the contents of the previous block together with the transaction data, and the result is the next block, which it then sends out to the network.

Now we have a way for nodes to verify the order of blocks: by verifying their hashes. Since the hash of any block depends uniquely (or very nearly

uniquely) on the previous block, a given set of blocks can only be verified in their true order.

This means we can order all transactions, and thus Irene can detect Sam trying to double-spend their Bobcoin on empty calories. Success!

Well, nearly. There's still a problem: what if two nodes independently compute and send out different versions of the "next" block? Won't the chain then split into two independently-evolving versions?

To avoid that, a kind of consensus voting takes place. If a node receives two different candidates for the "current" block, it keeps both of them until it can decide which is the authoritative block.

Doomed stubs

How can the node decide which block wins this race? Well, sooner or later it'll receive *another* block from the rest of the network. This will be "chained" to one or other of the two blocks.

In other words, the new block will be verifiable only against the hash of either Block A or Block B. Whichever it is, that's what the network considers to be the authoritative block, and the node can safely discard the other one.

This doomed "stub" version of the chain might still grow for another block or two, depending on the speed of propagation, but eventually it will be discarded. When faced with two competing versions of the chain, nodes will always opt for the longer one, because it represents the majority opinion that this is the "correct" chain. More work has gone into it, so the network votes with its feet—or rather, with its hashes.

Verifying a given transaction, then, is simply a matter of looking at the block that contains it. The more valid blocks that follow this one, the more confident we can be that the transaction has been accepted by other nodes.

Since hashing takes some non-zero amount of computing power, it's in a node's interest not to waste time, heat, and money by computing hashes for

a doomed stub. Thus, the network rapidly converges on a single authoritative ordering of Bobcoin transactions.

To repay nodes for their hard work in keeping the blockchain going, the node that successfully creates the next block earns a small fee, made up of the leftover fractional bits of Bobcoin from each transaction. Each node's version of the next block includes a transaction in which this fee is paid to their account. Thus, their reward for winning the race is to have their fee-earning transaction validated by the rest of the network.

Integrity

In a decentralised scheme like this where each node marks its own homework, isn't there a danger of fraud?

Surely all a malicious node has to do is compute the next block, based on a bogus transaction ("Alice pays Mallory one million Bobcoin", let's say), and then keep this stub chain going by computing all future blocks based on it (plus the subsequent genuine transactions). What's to prevent this?

Consensus

First, transactions are *signed* by the person who makes them, and we'll talk more about this in the next chapter. Suffice it to say for now that only Alice can create a transaction where Alice pays money to someone else, and anyone can independently verify it.

Second, hashing is hard work. It costs some amount of money to compute the hash of each block, so the longer this "fake" chain is kept up, the more it costs Mallory. Eventually it'll cost them more than they could steal with the fake transaction.

In effect, Mallory is fighting the combined hashing power of all the other nodes in the network, which are collaborating to keep the genuine chain going.

The genuine chain will rapidly become longer than the fake one, because it simply has more hashing power behind it. Very soon, all nodes will discard the shorter chain as invalid, and Mallory's efforts will be for nothing.

Sure, Mallory could recruit other nodes into their Bobcoin-stealing gang, but they'll still be defeated by the honest nodes. Only if Mallory can commandeer more than half of the total *hashrate* in the network can they hope to subvert the blockchain.

Such a *51% attack* is expensive and infeasible for large networks, which means that the more nodes join a given blockchain network, the more secure it is: safety in numbers.

Proof of work

A third way in which malicious blocks are discouraged is to make the hashing process as slow and expensive as possible, rather like the password storage hashes we looked at in the previous chapter.

In the case of the real-life Bitcoin network, for example, the hashing problem is made harder by requiring *miners*—nodes that want to produce blocks—to produce a hash value within a specific range.

For example, suppose you're given some transaction data and asked to hash it so that the 256-bit hash value is less than, say, a million. Of course you can only do this by mixing in some extra data of your own—some *nonce*—to produce the required value. And since you can't reverse the SHA-256 algorithm to find the input number you need (it's *preimage-resistant*), the only way you can do it is by guessing.

It's like a kind of benign brute-force attack, or a version of the “guess the number I'm thinking of” game for extremely patient players. You just keep guessing a different value for the nonce and hashing the block until, by pure chance, your hash value comes out as some number less than 1,000,000.

That might take a while: given the size of the 256-bit hashspace, your odds of producing an acceptable hash are about one in 10^{71} for each attempt. It therefore costs quite a bit of money just to produce each block, making

attempted fraud less attractive. Mining fees are structured so as to make it more rewarding to mine genuine blocks than fake ones.

In Bitcoin, the required *difficulty target* is a little easier than our example: at the time of writing, the chances of producing an acceptable hash are currently about one in 10^{64} . The difficulty target gradually increases over time, to keep pace with the expected increase in computing power.

Up in smoke

If I tell you that, despite this astronomical difficulty, a new Bitcoin block is mined, on average, about every ten minutes, that gives you some idea of the enormous computing resources devoted to the network.

Tens of thousands of Bitcoin nodes are all hashing as fast as they can, five hundred billion billion hashes a second, trying to guess a nonce that will produce a valid hash, and only one node can win that race. All the work done by the losing nodes is wasted, and simply goes up in smoke. Literally, since most of the electricity they use comes from burning fossil fuels.

This *proof of work* scheme, then, depends on wasting a massive amount of precious energy. Bitcoin alone consumes over 120 terawatt-hours of electricity every year, which is more than many countries.

In effect, Bitcoin adds another energy-hungry country to the world, and we don't really need another one of those.

It seems a shame that the most powerful distributed supercomputer in human history is being wasted on polluting our atmosphere and heating up the planet by solving math problems that are totally useless to humanity.

Proof of stake

Proof of work, then, is perhaps the worst idea ever, but fortunately there are some alternatives in prospect.

One is so-called *proof of stake*. In this scheme, rather than the most trusted nodes being the ones that have burnt up the most energy, they are simply the ones who have the most *money*. These nodes, the reasoning goes, have the biggest stake in the network, so they're least inclined to undermine it.

That's true to *some* extent, but those who find the democratising nature of cryptocurrency appealing may think that this is just another way of rigging the system in favour of its richest participants. And we have that already: it's called "capitalism".

Clearly we need a fair, efficient way of preventing fraud and double-spending that doesn't literally set fire to the planet, and we don't have that at the moment. Maybe someone will have a bright idea about this soon. Maybe it'll be you.

17. Authentication

Cryptography is the art of transforming information security problems into key management problems.

—Soatok Dreamseeker, “[Database Cryptography Fur the Rest of Us](#)”



In the chapter on hashing, we saw how a secure hash algorithm such as SHA-256 can be used to create a cryptographic digest of a message.

That's a good start, but for Alice and Bob to have a reliable way of exchanging secure messages, we'll need to do a little more. Let's see why.

Message integrity

The hash digest, or simply “hash”, gives Bob an independent check on the integrity of Alice's message. He can compute the hash value of the message for himself, and compare it with the one Alice attached. If they disagree, then the message has changed since Alice sent it.

Maybe it was modified by Mallory, or maybe a few bits just got flipped in transit by an errant cosmic ray. Either way, at least Bob can detect that something's gone wrong, and ask Alice to re-send her *billet-doux*.

But if the hash values agree, Bob can be pretty confident that the message is the same one that Alice sent, even if it's sent in clear. There's no way for Mallory or anyone else to modify the message in such a way that it hashes to the same digest, without carrying out a brute-force preimage search.

Hash preimage attacks

While a good hash algorithm, as we've seen, does everything possible to make this brute-force attack hard, it's still just a matter of time. If Mallory has a lot of resources to throw at the problem, they can eventually find a modified message that produces the same hash as Alice's original. They might get lucky and find a hash collision on the first try, or it might take years, but it *can* be done, and there's no way to prevent it.

Or is there? What if Alice also *enciphers* the message, and then sends Bob the ciphertext, accompanied by its hash? Won't that frustrate Mallory's evil plan?

Well, partly. If the hash value is sent in clear, Mallory can still brute-force a ciphertext that produces the same hash, which isn't ideal.

Sure, that ciphertext may not be decipherable at all by Bob, or it might produce a meaningless plaintext. That's not as serious as if Mallory could send a *meaningful* fake message, but it's still annoying, and interferes with Alice and Bob's communications.

Hashing was supposed to help us with that problem, so clearly there's a little more work to do.

Chosen ciphertext attacks

What if Alice enciphers, not just the plaintext, but the plaintext *plus its hash*? For example, suppose she starts with the message "I love you, Bob", computes its hash, appends it to the message, and then enciphers the whole thing?

Well, that's okay on the face of it, but it opens Bob up to something called a *chosen ciphertext attack*. This is where Mallory forces Bob to decrypt a number of ciphertexts of their choosing, and by doing so can gain some knowledge about the key.

We don't need to know the exact details of how chosen ciphertext attacks work against particular cryptosystems. It stands to reason, though, that if Mallory can send an unlimited number of messages, and Bob keeps deciphering them, eventually something bad will happen. In fact, this is known as the [Cryptographic Doom Principle](#).

To prevent this, Bob should never even try to decipher any message that doesn't have a valid hash. And that implies that the hash digest must be readable without any decryption.

If a cipher is like an envelope for a letter, then the hash should go on the outside of the envelope. That way, Bob doesn't even have to open the envelope to know that the message is bogus.

But now we're back where we started, with Mallory able to brute-force bogus messages with valid hashes. How can we prevent this, and give Bob a way to verify that the message really comes from Alice, without also making him vulnerable to a chosen ciphertext attack?

MAC

One neat idea is for Alice to hash, not just the plaintext, or even the ciphertext of her message on its own, but the *ciphertext plus the key*. Then it's impossible, or at least very difficult, for anyone without the key to even *verify* the hash.

That's because in order to verify the hash, Mallory needs to compute it, and that means they need the whole text that Alice used to compute hers. But they don't *have* this, because Alice, naturally enough, doesn't include the key in her message.

Bob, of course, has the key, so he can check the hash very easily. All he has to do is append his own copy of the key to the ciphertext, compute the hash,

and compare it with the one Alice sent.

This clever scheme is called a *Message Authentication Code* (MAC). Even though Mallory could still brute-force a valid preimage for Alice's hash, it won't fool Bob, because when he appends the key to Alice's ciphertext and hashes the result, it won't match.

And Mallory can't generate a bogus message that produces the required hash *when the key is appended*, because they don't have the key. Checkmate?

Well, not quite.

Length extension attacks

Hashing the ciphertext and key together is much better than hashing just the ciphertext, but it's still not perfect. Some hash algorithms, including SHA-256, are vulnerable to what's called a *length extension attack*.

This loophole means that, while Mallory can't alter the original message without being detected, they *can* append extra text to it. As you can imagine, this could be problematic.

Alice's original message might be "I love you, Bob", and Mallory could append some text to it like this: "...but for reasons I can't go into, our love can never be. I'm leaving town tonight, goodbye forever." Even without knowing the key, they can use Alice's original hash plus the new text to produce a new, valid hash.

When Bob checks the hash, it looks to him as though this is a genuine message. Oh no! Bob and Alice's love-boat has been torpedoed before it even really got under way. And all because Alice was a bit careless with her message discipline.

HMAC

We can't let that happen, so we'll need to make a slight, but significant, improvement to our scheme. We'll have Alice hash the "key + message"

combination, as before, but we'll also add an extra step: hashing the key *plus the hash*.

Let's take that step by step:

1. Alice joins the key and ciphertext and computes the resulting hash (called the *inner hash*).
2. Then Alice joins the key with the inner hash and uses the result to compute the *outer hash*.
3. Finally, Alice sends the ciphertext along with the outer hash.

If our hash function is H , let's say, then a simple hash of the message would be $H(\text{message})$. We know that's no good, for the reasons we already discussed.

A simple MAC, then, would be $H(\text{message} + \text{key})$, or $H(\text{key} + \text{message})$. These are better, as we've seen, but still vulnerable in certain circumstances.

So the improvement we're making is to compute $H(\text{key} + H(\text{key} + \text{message}))$ instead. This seems weird, but it's cryptographically sound. The inner hash is obscured, so Mallory can't use it to carry out a length extension attack. The outer hash still requires the key, so Mallory can't just brute-force a preimage.

This scheme is called a *Hash-based MAC* (HMAC). It can give a message authenticity even without secrecy, which is handy.

Suppose Alice wants to send a message to the whole world proclaiming her love for Bob (love tends to have that effect, after all). She can put it on her website, in clear, but also attach an HMAC of the message.

Only Alice could have generated that HMAC, and only Bob can verify it, because they are the only two people who have the key. How satisfying!

Key exchange

Actually, that brings us to another problem, and one that we haven't really talked about so far in this book. Every encryption or authentication scheme we've looked at has relied on exactly this point: Alice and Bob must share a copy of the key. But how, exactly, is that to be arranged?

The problem

Suppose Alice and Bob decide that they want to communicate securely. It's easy enough for Alice to generate a secure key, especially if she's read the earlier chapters of this book, and has access to a good source of cryptographic entropy. But how is she now to get a copy of that key to Bob?

Meeting in person would obviously be the best way: she can hand over a physical device such as a flash drive with the key stored on it, or even just write it down on paper.

But suppose Alice and Bob are too far apart to meet in person, or at least that it's not convenient for them to meet every time they need to exchange a new key. What could they do instead?

Alice could just email the key to Bob, but we know that's no good: Eve might intercept the email. Worse, Mallory might replace Alice's key with one of their own, and then they'd be able to read and modify all of Alice and Bob's cipher traffic undetected.

The same concerns apply to a phone call, letter, or any other non-secure way of communicating.

Key splitting

If she's wise, Alice won't trust the entire key to a single channel. Instead, she'll split it up into multiple parts, and send each one a different way: half by email, for example, and half by phone. That makes Eve and Mallory's job harder, which, as you know, is all we can ever hope to do, but it's well worth doing all the same.

However, if at all possible, Alice would really like to avoid having to send *any* part of the key over an insecure channel.

Of course, if Alice and Bob already had a *secure* way of communicating, they wouldn't need to exchange a key in the first place! So we face a seemingly unresolvable problem. The only way for them to communicate securely is to share the key, but they can only do that by communicating securely. What to do?

Well, when you put it that way, the answer's obvious, isn't it? *Don't share the key.*

Asymmetric encryption

Okay, that might need a little more explanation, so here goes. Suppose there were a way for Alice to generate a special kind of key, one that comes in two parts. One part is secret, and Alice keeps this to herself, as with an ordinary key.

The other part of the key, though, doesn't need to be secret. In fact, it can be *public*. She could send it to Bob via some insecure channel such as email, or just put it on her website.

Now Bob—or anyone else—can use the public part of Alice's key to encipher a message in such a way that only Alice can read it. That is, only someone in possession of the *private* part of the key can decipher a message encrypted using the public part.

The neat thing is that Alice and Bob never have to meet, or communicate any secret information over a potentially insecure channel. Yet they can still have a securely-encrypted conversation. So how does *that* work? Is it even possible?

As we saw in the first chapter, the history of cryptography is long: people have been exchanging hidden messages for thousands of years, at least as long as writing has existed, and perhaps before that. But until very recently, they've always had to rely on being able to communicate keys between the

sending and receiving parties, and that creates an obvious vulnerability for an attacker to exploit.

This is known as the *key exchange problem*.

Public and private keys

All the cryptography we've discussed up to now has been what's called *symmetric*: the symmetry here is that the enciphering and deciphering processes use the same key.

If there were an alternative scheme where you could encipher with one key, but decipher with a different (yet somehow related) key, that would be called *asymmetric* encryption. Wouldn't that be nice?

Well, as you probably guessed, I'm about to outline just such a scheme. Because the point of it is for one of the keys to be public, it's also called *public-key* cryptography, which I'm sure you've heard of. Now let's see how it works.

It's a bit like one of those magician's tricks where you think of a number, they tell you to multiply or divide it a bunch of times, and when you give them the result, they miraculously tell you the number you first thought of.

It's not really magical, of course: it's *mathemagical*. The starting and ending numbers are related in a completely deterministic way, just one that's not particularly obvious.

The Diffie-Hellman-Merkle protocol

Suppose you could do this kind of trick with a private key; it's just a number, after all. Scramble it in some non-trivial, non-reversible way, and you end up with another number that you can safely reveal to the world: the public key. (Actually, that sounds a lot like a hash, doesn't it?)

And suppose further that if a message were enciphered with this public key, you could only decipher it using the corresponding *private* key. So what

mathematical function could we use to scramble keys in this sort of system?

It would have to be a one-way function, similar to the secure hash functions we've discussed already in this book, though with some different specific properties. Let's look at one such system, named Diffie-Hellman-Merkle (*DHM*), after its inventors.

While DHM is not a full public-key cryptosystem in itself, it does provide a way for two parties to communicate in public to arrive at a shared private key, without revealing what it is.

If Alice and Bob can do this easily, then they can generate and use a new key for every conversation (this is known as *forward secrecy*). Now, even if Eve or Mallory can intercept a message and use brute force or cryptanalysis to break the key, the damage is limited. They can read that one message, but it won't help them decrypt future messages, which will use new keys.

A one-way function for key exchange

The particular asymmetry, or one-way-ness, used by DHM is that between the respective difficulties of computing *exponents* and *logarithms* (set your calculator to "maths" for the following section).

Exponentiation is when you raise a given number to some power: for example, 2^{10} . Here, 10 is the exponent, and to calculate the result we multiply 2 by itself 10 times: the answer is 1024. Not too hard: you can do that with pencil and paper, or a few carefully-arranged transistors.

On the other hand, computing the logarithm is the inverse operation: you start with 1024 and ask how many times you have to multiply 2 by itself to get this result. The answer is known as the *logarithm* of 1024 (to *base 2*). It's 10, of course, but if we didn't already have that information, we'd have to work it out by trial and error.

Finding logarithms, then, is a much harder problem than exponentiation. It can't be done *analytically*: that is, there's no formula that just gives you the answer in a single computation.

Instead, you have to do it iteratively: guessing the answer, raising 2 to that power and seeing how close you got to the required number, then successively improving your guess until you think it's good enough.

Sounds a lot like a brute-force attack, doesn't it? And that's exactly what it is.

In just the same way, enciphering a block with a given key is fast, but reverse-engineering a ciphertext when you don't know the key is extremely slow. And, while hashing a block with SHA-256 is fast, especially with a custom Bobcoin mining rig, guessing the exact preimage that produces a desired hash is much slower, as we saw in the previous chapter.

A simplified, though inaccurate, way to think about the DHM key exchange scheme is like this: imagine Alice generates a private key and sends Bob the hash of it. Now imagine Bob generates his own key and sends Alice the hash of *that*.

Now add a little hand-wavy mathematical fairy dust, like the after-dinner magician, and it turns out that Alice and Bob can both independently use these different hashes to calculate the *same* number.

That number can now be the private *session key* they use to communicate: perhaps they'll arrange to meet for drinks, and maybe take in a magic show afterwards. Who knows where such an enchanted evening might end?

A DHM worked example

Okay, the hand-waving was a bit quicker than the eye there, so let's go through the process in more detail with an example.

First, Alice and Bob agree on a pair of numbers they'll use for this calculation, called the *modulus* and the *base*, and these can be public. Let's say they use a modulus $p = 13$ and base $g = 6$. Not just any two numbers will do, but never mind: these will work for the example.

The next step is for Alice to generate a secret number a that only she knows, perhaps using a random process. We'll use 5 as an example, but a

really secure private key would be much larger, as you know. Using a small number makes the sums easier for explaining the idea, though.

Alice now applies her one-way function (exponentiation) to compute the result she can send publicly to Bob. Specifically, she computes:

$$g^a \bmod p$$

In other words, she raises the pre-agreed base number g to the a -th power and takes the result modulo p (remember the modulus operation? Of course you do.)

Substituting in our chosen values, she gets:

$$6^5 \bmod 13 = 2$$

Now over to Bob, who generates his own secret number $b = 4$, and then munges it in the same way that Alice did. That is, he raises the base g to the b -th power, mod p . He gets:

$$6^4 \bmod 13 = 9$$

Alice and Bob can now exchange these results in public, quite safely. No one can use them to work backwards to the corresponding secret numbers, except by brute-force guessing.

So now Alice knows that the result of Bob's calculation is 9, and Bob knows that the result of Alice's calculation is 2. Take a deep breath, because we're almost there.

The reveal

Here comes the final step. Alice does another modular exponentiation:

$$B^a \bmod p$$

That is, she raises Bob's result (9) to the power of her own secret number (5), mod 13. So:

$$9^5 \bmod 13 = 3$$

This answer is the shared session key that Alice and Bob will use to communicate. So, if the DHM scheme works, Bob should be able to calculate the same number by a similar process ($A^b \bmod p$). Let's see:

$$2^4 \bmod 13 = 3$$

Now *that's* magic! Or rather, it isn't, but it seems magical if you don't know how the trick works. It relies on the not particularly surprising fact that:

$$(g^a)^b = (g^b)^a$$

In other words, it doesn't matter in which order you do the two exponentiations: a first, or b first, you get the same answer. For example, if you square 2 and then cube the result, you get 64, and if you instead cube 2 and then square the result of *that*, you still get 64.

As we saw earlier, real DHM key exchanges would use much larger numbers, and there are a few more fiddly details that we needn't worry about, such as how to choose the base and modulus. You get the basic idea, though.

For more information, please reread, or have a look at Joumana El Alaoui's kgen project, which implements the DHM protocol in Go:

<https://github.com/joumanae/kgen>

Public-key cryptography

As neat as it is, DHM, as we've seen, isn't by itself a true public-key cryptosystem. Alice and Bob are still using plain old symmetric encryption, meaning they have to share a key.

It's just that we've somewhat streamlined the process of *exchanging* that key, meaning that they can safely use an insecure channel to communicate

about it. But they still have to *have* that communication before they can start an encrypted conversation.

By contrast, with a public-key cryptosystem, Alice could send Bob a private, encrypted message without any preamble. She wouldn't have to arrange with Bob to do the Diffie-Hellman-Merkle dance, or leave a copy of her key behind a loose stone in the garden wall, or anything like that.

Instead, she could just look up Bob's public key online, in the cryptographic equivalent of the phone book, and send him a message right out of the blue. Yet, only Bob (or someone who's managed to steal his *private* key) could read it. So how would *that* work?

RSA

Again, there are many different ways of doing this, but we'll take a brief look at one fairly well-known scheme. It was invented first by Clifford Cocks, and then again independently by Rivest, Shamir, and Adleman, which is why it's now known as *RSA*.

RSA relies on the same sort of one-way function trick as DHM, and it works like this. Alice chooses two large, random prime numbers, puts them into a mathemagical sausage machine, and out pop two related numbers: her *private key* and her *public key*.

Alice can quite safely distribute her public key, or register it with some public directory, and anyone can use it to send her an encrypted message. However, only Alice can decrypt and read it. No one else, not even the sender, can unscramble the message.

In practice, though, it's usual to encrypt the message to your own public key as well as the intended recipient's, so that you can read messages you've sent, and RSA allows for that. Indeed, you can encrypt a message to as many recipients as you want, including yourself. All you need are the corresponding public keys.

So far, so straightforward. The bit that makes it seem like magic is the relationship between the private and public keys. They must *be* related, or

the encryption wouldn't work, naturally. But they mustn't be so *obviously* related that someone with the public key can use it to reverse-engineer the private key.

That's the whole trick, and it depends on the fact that these two numbers have a one-way relationship. Just as finding logarithms is harder than exponentiation, while it's easy to multiply two numbers together, *factorising* the result (working backwards from it to find the two numbers) is much harder. (At least, for now. We'll return to this point later.)

Signing

This is great, but it still doesn't quite solve the problem that we faced earlier in this chapter: how can Bob know that a given message actually comes from *Alice*, and not merely someone claiming to be Alice?

In a way, public-key cryptography such as RSA makes this problem worse, not better. At least with symmetric encryption, the key is secret, and so Bob knew that a message encrypted with it could only have come from someone who shares the secret. Now that Bob's key is public, though, the message could have come from anyone! How can Alice prove that it's really from her?

Well, we can solve this *signing* problem by exploiting the connection between the private and public keys in a different way.

We've already seen that Alice can use her private key to decrypt a message sent to her by anyone in possession of her public key. But, conversely, she can also use the same private key to *encrypt* a message that anyone in possession of her public key—that is, everyone—can *read*.

What would be the point of doing this, and what message would she encrypt in such a way? Well, perhaps you've already guessed.

Alice can compute a hash digest of her message, and encrypt this hash with her private key. Anyone who has her public key can decrypt and verify this hash, so they know that the message it's attached to could only have come from Alice.

An RSA-style private key serves two purposes, then: first, people can send you messages that only you can read, and second, you can send people messages that only you could have *sent*. This is very neat.

Authentication

In essence, cryptographic authentication consists of two separate, but related problems: *integrity* (is the message intact?) and *signature* (who sent it?) HMAC solves the integrity problem, while RSA and other public-key cryptosystems can solve the signature problem.

Indeed, you could think of RSA message signing as a kind of extension of the HMAC idea, but where HMAC uses a symmetric key, RSA is asymmetric. It's also rather slow, because there's a lot of crunching of some very big numbers.

In practice, then, RSA is often used in a similar way to Diffie-Hellman-Merkle: as a way to set up an initial session key that will be used for a subsequent, symmetrically-encrypted conversation.

That's how the encryption in your web browser works, for example: Transport Layer Security (*TLS*, the modern protocol which succeeded the older Secure Sockets Layer, or *SSL*) uses asymmetric encryption. It's not necessarily DHM or RSA, but those are among the options.

During the *handshake* phase of the connection, the client and server use asymmetric encryption to negotiate the session key, and in turn that key will be used to symmetrically encrypt the rest of the conversation.

Verification

Now, you might be wondering: “If Alice and Bob have never met or communicated before, how can Alice know that Bob’s public key is really *Bob’s*, though?”.

For example, what if Eve decided to surreptitiously register her own public key under Bob’s name in the global directory? Then she could read any

message intended for Bob and encrypted to that key. What guarantee does Alice have that this hasn't happened?

That's a good question, and a lot can depend on the answer. In the TLS example, I can have an encrypted exchange with some website—`bankofbob.com`, perhaps—and be reasonably confident that no one other than me and the owner of that website are privy to its contents.

But, to your point, how can I know if that's the *real Bank of Bob website*? After all, it's easy to register a domain, and even generate a TLS certificate for that domain. What's to stop Eve doing just that?

Well, nothing. What I—or rather Alice—would really like to do is meet Bob somewhere in person, perhaps over a crisp, chewy, wood-fired pizza, and a glass or two of the amusing house wine. There, they can verify each other's identity, and make a close inspection of each other's keys, and anything else of interest.

Then, in future interactions, Alice could be confident she's really talking to Bob, and vice versa.

But we already know that if Alice and Bob had a secure way to communicate about their keys, then they wouldn't need the keys in the first place. So, while a pizza date would be delightful, it doesn't solve our key exchange problem.

The chain of trust

To see how to get around this, suppose there's some third party that both Alice and Bob trust: we'll call him Trent. If Trent says that Bob's public key is really Bob's, that'll be good enough for Alice. She, or anyone else who trusts Trent, can check with him to verify a public key that claims to be Bob's.

In the TLS example, Trent would be the *certificate authority* (CA): he can sign Bob's certificate with his own, thus verifying that it's really Bob's. Anyone who trusts Trent's certificate should also trust Bob's certificate.

In fact, to extend this idea, and to reduce the workload on Trent of signing every TLS certificate in the world, Trent could delegate his trust to others. He could sign a bunch of people's certificates, making them all certificate authorities, and Alice or Bob can choose which CA they'd prefer to deal with.

That choice doesn't matter, except that some will be more expensive than others, especially if you pay for the super-fancy TLS certificates with the gold edging. The point is that all the CAs are equally trustworthy, because Trent trusts them all equally, and if you trust Trent, you implicitly trust them too.

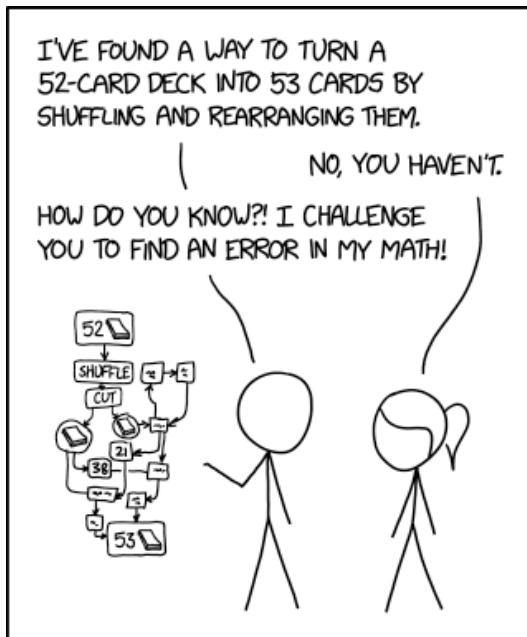
In fact, you can extend this *chain of trust* as far as you like. As long as you can trace the signing chain for any given certificate all the way back to Trent—the *root CA*—then it's just as good as verifying it directly.

It seems like all the pieces are falling into place now. With secure keys, strong encryption, reliable authentication, and scalable key exchange protocols, we've got everything we need to do some *real* cryptography. Let's try.

18. Cryptography

Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.

—Bruce Schneier, ["Memo to the Amateur Cipher Designer"](#)



EVERY CONVERSATION BETWEEN A PHYSICIST
AND A PERPETUAL MOTION ENTHUSIAST.

You should never try to invent your own cipher, or, for that matter, a perpetual motion machine. In both cases the results are likely to be disappointing, and it's not a good use of your time.

We've learned a lot in this book about how ciphers work, or don't, principally by *trying* to invent them ourselves, but now you might be wondering what the upshot of it all is, in practical terms.

I mean, if you need to encrypt stuff in a Go program in real life, you definitely shouldn't use the shift cipher, or any other simple cipher that you invented (or read in the Knowhow Book of Spycraft). You shouldn't even

write the code to implement some existing cipher yourself. What *should* you do, then?

That's easy, and I can give you the answer in three words. *Just use AES.*

A little history

The initials “AES” stand for “Advanced Encryption Standard”, as perhaps you know. To explain what’s so advanced about it, we’ll need to say a word or two about the history of previous encryption standards.

In the beginning was the Caesar cipher (actually, he stole the idea from the ancient Egyptians). Fast-forward through two thousand years of fascinating cryptographic developments, which I urge you to look into after class, including Vigenère ciphers, the Enigma machine, and one-time pads. It’s fun stuff, but we can’t stop to talk about it: we’ve already arrived in 1973. Sorry about the bumpy landing.

The origins of DES

It makes a lot of sense for a government, or any other big organisation with secrets, to have an official standard cipher. The United States government is pretty big, and has a lot of secrets (not always terribly well-kept), so in 1973 it invited proposals for just such a national standard encryption system.

The result was the Data Encryption Standard (*DES*). DES uses a key size of 56 bits. Yes, that sounds rather small, especially to those who’ve read this book, and so it is. The restricted key size was imposed by the National Security Agency (NSA), the US cryptographic intelligence bureau, for predictable reasons: they didn’t want anyone using a cipher that they themselves couldn’t break.

You’d think governments would be in favour of encryption that’s as strong as possible in order to keep their own secrets safe, and so they are, up to a point. The problem is that they also want to know *other* governments’ secrets. The NSA’s preferred cipher, then, would have a key short enough

that *they* can break it, with their tremendous resources, but also long enough that other, less well-equipped spy agencies can't.

The chosen length of 56 bits thus tells us something about the brute-force cracking abilities of the NSA in the early 1970s, and by extension about those of its rivals (both foreign and domestic).

However, the 1970s were a long time ago, as I'm increasingly aware, and as an old cryptographer's adage has it, "Attacks always get better; they never get worse."

Tripling down on DES

So, by the mid-1980s, concern about practical brute-force attacks on DES had become widespread, and the quick-fix solution, for really *important* secrets, was to use "Triple DES". As the name suggests, it simply consists of applying DES three times.

This doesn't sound all that promising on the face of it: three weak passwords are not much better than one, for example. It's clearly a case of kicking the can down the road, but governments have a bad habit of doing just that, as you may have noticed.

Something new would soon be needed.

AES

By the late 1990s, brute-forcing a DES key within a few days was possible on a machine costing a couple of hundred thousand dollars. This is cheap enough, relatively speaking, to make any government nervous. Once DES had fallen, indeed, could Triple DES be far behind? So proposals were invited for a new standard, to be called the *Advanced Encryption Standard* ("This time, we'll get it right!").

Rijndael

The cipher selected was one named Rijndael, after its inventors Vincent Rijmen and Joan Daemen. While Rijndael can work with a range of block sizes, the AES standard itself specifies a fixed block size of 128 bits (16 bytes). Similarly, while the underlying algorithm supports many different key sizes, AES limits the selection to either 128, 192, or 256 bits.

We won't bother with the shorter key sizes (nor should anyone else), so let's assume from now on that we're talking only about AES-256, which is to say Rijndael using 256-bit keys. Rijndael is a very strong and sophisticated cipher, so using it with short keys would be silly, like owning a Ferrari but driving it everywhere in first gear.

Now, you don't actually need to know anything about how AES works in order to use it, as long as you follow the usual good cryptographic practices like choosing secure keys, and so on. But you're *here*, and it's interesting, so let's take a quick overview of the internal structure of AES, at a functional level.

You already know how a block cipher works in general, because you've implemented one. And AES works the same way: it splits the input plaintext into equal-sized blocks, and then combines each block with the key. The result is the ciphertext.

And, just as with any block cipher, AES can be used in a variety of operating modes, some of which you'll also recall from previous chapters: ECB (bad), CBC (good), and so on.

We'll talk more about operating modes for AES shortly, but let's first see *how* the key is combined with the plaintext, specifically. That's the clever bit, of course, so it might take a little explaining.

The starting grid

As we've seen, AES uses a block size of 16 bytes, which is shorter than we used in the shift cipher. However, it's a nice round number, or rather a nice square number: 4×4 .

That's not a coincidence, because we're going to arrange the 16 bytes of block data in a 4×4 grid. Instead of writing them in rows, left to right, we'll write them top to bottom, in so-called *column-major* order, like this:

01	05	09	13
02	06	10	14
03	07	11	15
04	08	12	16

Why use a grid, you ask? Well, it's a bit like one of those puzzle games where you have to shuffle the tiles around to form a sentence or reveal a picture. We're going to shuffle the rows and columns around in a similar way to obscure the plaintext before encrypting it.

In our shift cipher, we encrypted each block using a single step: we combined each plaintext byte with the corresponding key byte, in our case by adding them together ($\text{mod } 255$). AES does something similar, but it repeats the procedure many times, using multiple successive *rounds*.

Round and round

In AES-256, there are fourteen rounds, and before we even start encrypting any blocks, we generate a separate 128-bit key for each of these rounds. All these *round keys* are derived from the 256-bit cipher key itself, by a complicated mathematical process we'll glide smoothly over here.

Once we have the round keys, we can begin encrypting. In the first round, we simply combine each plaintext byte with the corresponding byte of the round key (let's call this phase “AddRoundKey”).

Despite the name, rather than adding the two bytes together (as the shift cipher does) AES uses the *XOR* operation. Just like addition, this is a fast, simple, reversible way of combining two bytes. As a bonus, it's very easily implemented in hardware.

The output grid from the first round now becomes the input to the second round, and so on. In this, and each of the subsequent rounds before the last, we'll again combine the data with the corresponding round key, but we also perform three extra steps beforehand (I did warn you it was complicated).

Confusion and diffusion

In the first of these phases, “SubBytes”, we substitute every byte of the block data with a different byte, obtained from a lookup table called the *S-box*. This is just a fixed table with 256 entries, mapping every possible input byte to some different output byte. All AES implementations use the same S-box table, and it’s published, so the enemy has it too, but that’s okay. Its purpose is just to make it harder to reverse-engineer the key from the ciphertext.

The next phase is called “ShiftRows”: we shift the rows horizontally, like the sliding tile puzzle. We leave the first row as it is, then we shift the second row by one place to the left, the third row by two places, and the last by three places. The overall effect of this is to make sure that each column in the output contains a byte from each of the four input columns.

Why do this? Good ciphers obscure the plaintext in two main ways, known in the trade as *confusion* and *diffusion*. Confusion involves changing the value of each byte, as in the previous SubBytes phase, and we did that in the shift cipher too.

Diffusion, on the other hand, involves moving those bytes around *within* the block, which the shift cipher doesn’t do, but AES does. The point, again, is to make it even harder for Eve to work backwards from the ciphertext to the plaintext.

It’s just like the simple “rail fence” cipher we find in the KnowHow Book of Spycraft. For example, we might arrange the phrase “ATTACK AT DAWN” in a column-major grid as follows:

A	C	T	W
T	K	.	N
T	.	D	.
A	A	A	.

The resulting ciphertext, reading off each row in turn, is “ACTW TK.N T.D. AAA.”, and all the recipient has to do is write it out in a similar 4×4 grid to read off the plaintext in columns. Diffusion on its own, of course,

isn't terribly secure, which is why it needs to be combined with confusion to produce an effective cipher.

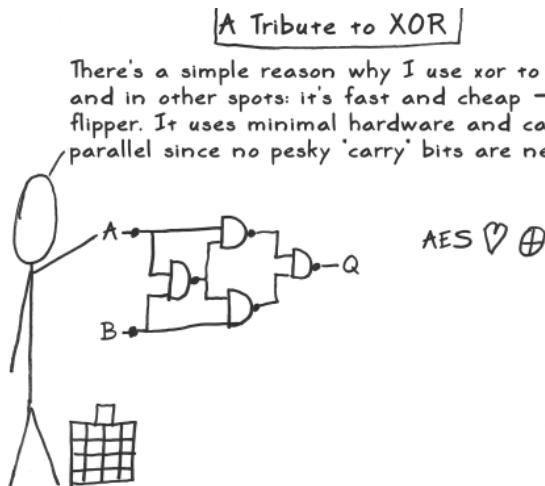
The third phase of AES encryption, “MixColumns”, is also a diffusion step, but operating on columns instead of rows. It applies a matrix transformation to each column such that each byte in the output column depends on all four input bytes of that column. Again, the details don't matter for this discussion, but you can take it for granted that they're mathemagical.

Putting it all together

Finally, we apply the round key, and the whole process is then repeated in the next round. This continues until the last round, which is the same as the others except for the omission of MixColumns. So here's the recipe:

- Round 1: AddRoundKey
- Rounds 2-13: SubBytes, ShiftRows, MixColumns, AddRoundKey
- Round 14: SubBytes, ShiftRows, AddRoundKey

And that's it. It *sounds* a bit complicated, and of course cryptographically speaking it is: that's the point. If you really want to dive into the details, read Jeff Moser's wonderful [Stick Figure Guide to AES](#):



However, as complicated as each of these phases are, they can be implemented very simply using a pre-generated lookup table. This is tremendously efficient, and it doesn't take much code. So, in practice, AES implementations can be remarkably concise, straightforward, and easy to read.

To prove that point, let's take a look at the implementation in Go's `crypto/aes` package.

Implementing AES

In fact, when you use AES encryption in Go (which we'll see how to do later on), you'll almost certainly be using the hardware-accelerated AES instructions provided by your CPU. Depending on your `GOARCH` setting, therefore, Go will select the appropriate assembly-language implementation for you automatically. These are fast, but *not* particularly easy to read, especially if you just want to get the gist of the algorithm.

Getting ready to rumble

There is a pure Go implementation, though, for architectures that don't have on-chip AES acceleration, so let's look at that. Here's the beginning of the `encryptBlockGo` function:

```
// Encrypt one block from src into dst, using the expanded key
xk.

func encryptBlockGo(xk []uint32, dst, src []byte) {
    _ = src[15] // early bounds check
    s0 := binary.BigEndian.Uint32(src[0:4])
    s1 := binary.BigEndian.Uint32(src[4:8])
    s2 := binary.BigEndian.Uint32(src[8:12])
    s3 := binary.BigEndian.Uint32(src[12:16])
```

([crypto/aes/block.go](#))

This is the equivalent of the `Encrypt` method we wrote for the shift cipher: it takes a block of plaintext (`src`), and enciphers it into the destination block (`dst`), using the set of expanded round keys `xk`.

Since AES blocks should be 16 bytes in size, the first thing we do is a quick sanity check: we reference `src[15]`. If `src` is too short, this will panic, which was bound to happen sooner or later anyway—we’re just getting it out of the way without wasting any further computation.

Then, we extract each of the four 4-byte columns of our grid, `s0` through `s3`, from the source block. These will be modified by the various rounds as we go along. For convenience, we’re dealing with each 4-byte chunk as a single Go `uint32` value.

Ding ding, round one

We’re now ready to start the first round, which as you recall consists just of `AddRoundKey`:

```
// First round just XORs input with key.  
s0 ^= xk[0]  
s1 ^= xk[1]  
s2 ^= xk[2]  
s3 ^= xk[3]
```

([crypto/aes/block.go](#))

Recall that `xk` contains the set of round keys that was expanded from the original key. Each round key is 128 bits, or 16 bytes, so it takes up four `uint32` elements in the `xk` slice. In other words, the first four elements of `xk` are the first round key.

The XOR operation is represented in Go by `^`, so this code updates each chunk of data by XORing its bits with the corresponding bits of the round key.

The diffusion loop

Next, since all the remaining rounds but the last will be the same, we can do them in a loop. But the number of rounds varies depending on the key size being used, so we first do a calculation based on the size of `xk`:

```
// Middle rounds shuffle using tables.  
// Number of rounds is set by length of expanded key.  
nr := len(xk)/4 - 2 // - 2: one above, one more below
```

([crypto/aes/block.go](#))

In other words, since each group of 4 elements of `xk` represents a single round key, we can find out how many rounds there are in total by dividing the length of `xk` by 4. Since the first and last rounds are special, we won't include them in this loop, hence the need to subtract 2.

Here's the loop, then:

```
k := 4  
var t0, t1, t2, t3 uint32  
for r := 0; r < nr; r++ {  
    t0 = xk[k+0] ^ te0:uint8(s0>>24) ^ te1:uint8(s1>>16)  
        ^ te2:uint8(s2>>8) ^ te3:uint8(s3)  
    t1 = xk[k+1] ^ te0:uint8(s1>>24) ^ te1:uint8(s2>>16)  
        ^ te2:uint8(s3>>8) ^ te3:uint8(s0)  
    t2 = xk[k+2] ^ te0:uint8(s2>>24) ^ te1:uint8(s3>>16)  
        ^ te2:uint8(s0>>8) ^ te3:uint8(s1)  
    t3 = xk[k+3] ^ te0:uint8(s3>>24) ^ te1:uint8(s0>>16)  
        ^ te2:uint8(s1>>8) ^ te3:uint8(s2)  
    k += 4
```

```
s0, s1, s2, s3 = t0, t1, t2, t3  
}
```

([crypto/aes/block.go](#))

Yes, it's a little dense, but it's written for efficiency, not readability. It combines SubBytes, ShiftRows, MixColumns, and AddRoundKey in a single operation, on each of our four chunks `s0` through `s3`.

We implement SubBytes by replacing each data byte by its corresponding byte in the pre-generated lookup tables `t0` through `t3`. By careful adjustment of indexes and some bit-shifting, we also implement ShiftRows and MixColumns. Finally, we XOR each of the resulting chunks with its corresponding part of the round key.

Fortunately, you don't need to understand every detail here to get the general idea: it's a bunch of lookups and bit-shifts. Indeed, we could have written out each of the round steps separately, to make it clearer what's happening at each stage, but the standard library authors understandably aren't concerned with that.

The last round

Here's the last round:

```
// Last round uses s-box directly and XORs to produce output.  
s0 = uint32(sbox0[t0>>24])<<24 | uint32(sbox0[t1>>16&0xff])  
<<16  
    | uint32(sbox0[t2>>8&0xff])<<8 | uint32(sbox0[t3&0xff])  
s1 = uint32(sbox0[t1>>24])<<24 | uint32(sbox0[t2>>16&0xff])  
<<16  
    | uint32(sbox0[t3>>8&0xff])<<8 | uint32(sbox0[t0&0xff])  
s2 = uint32(sbox0[t2>>24])<<24 | uint32(sbox0[t3>>16&0xff])  
<<16
```

```

    | uint32(sbox0[t0>>8&0xff])<<8 | uint32(sbox0[t1&0xff])
s3 = uint32(sbox0[t3>>24])<<24 | uint32(sbox0[t0>>16&0xff])
<<16
    | uint32(sbox0[t1>>8&0xff])<<8 | uint32(sbox0[t2&0xff])

s0 ^= xk[k+0]
s1 ^= xk[k+1]
s2 ^= xk[k+2]
s3 ^= xk[k+3]

```

([crypto/aes/block.go](#))

In the middle rounds, the S-box substitutions were already factored in to the pre-generated lookup tables which also included the effect of MixColumns. Here, since we're skipping the MixColumns step, we can refer directly to the `sbox0` table, which is just a constant.

Having thoroughly munged our data chunks `s0` through `s3`, we finally write them back to the output block `dst`:

```

_ = dst[15] // early bounds check
binary.BigEndian.PutUint32(dst[0:4], s0)
binary.BigEndian.PutUint32(dst[4:8], s1)
binary.BigEndian.PutUint32(dst[8:12], s2)
binary.BigEndian.PutUint32(dst[12:16], s3)
}

```

([crypto/aes/block.go](#))

Again, we check that `dst` is long enough before writing the enciphered block to it. So, that's it: the core of AES in just 45 lines of Go (if you don't count the constant lookup tables). Not bad!

Some of those actual lines do make your head hurt a bit, especially if you haven't had enough coffee, but that's okay. You don't need to *write* this code, just enjoy it as a miniature work of art.

AES encryption in practice

More to the point, of course, we can *use* it to do some actual encryption with AES. And that's extremely simple.

Enciphering with AES-CBC

Remember our encipher program using the shift cipher? Here's the relevant part of it:

```
block, err := shift.NewCipher(key)
...
iv := make([]byte, shift.BlockSize)
_, err = rand.Read(iv)
...
enc := cipher.NewCBCEncrypter(block, iv)
...
enc.CryptBlocks(ciphertext, plaintext)
```

([Listing shift/8](#))

And here's the same program, but encrypting with AES instead of the shift cipher:

```
block, err := aes.NewCipher(key)
...
iv := make([]byte, aes.BlockSize)
```

```
_ , err = rand.Read(iv)
...
enc := cipher.NewCBCEncrypter(block, iv)
...
enc.CryptBlocks(ciphertext, plaintext)
```

([Listing aes/1](#))

Almost identical! Which is the point, of course: the standard library interfaces make it trivially easy to plug and play with different ciphers in more or less the same way.

Let's prove that.

Updating encipher and decipher

GOAL: Go ahead and update the `encipher` and `decipher` programs to use AES instead of the shift cipher.

HINT: It's a good idea to make this a new project, since it's not really part of the `shift` package anymore. Feel free to borrow the `Pad` and `Unpad` functions, though, since we'll still need those.

SOLUTION: Here's my version of `encipher`:

```
package main

import (
    "bytes"
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
```

```
"encoding/hex"
"flag"
"fmt"
"io"
"os"
)

func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := aes.NewCipher(key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    plaintext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    iv := make([]byte, aes.BlockSize)
    _, err = rand.Read(iv)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    os.Stdout.Write(iv)
```

```

    enc := cipher.NewCBCEncrypter(block, iv)
    plaintext = Pad(plaintext, aes.BlockSize)
    ciphertext := make([]byte, len(plaintext))
    enc.CryptBlocks(ciphertext, plaintext)
    os.Stdout.Write(ciphertext)

}

func Pad(data []byte, blockSize int) []byte {
    n := blockSize - len(data)%blockSize
    padding := bytes.Repeat([]byte{byte(n)}, n)
    return append(data, padding...)
}

```

[\(Listing aes/1\)](#)

And, just for completeness, here's decipher:

```

package main

import (
    "crypto/aes"
    "crypto/cipher"
    "encoding/hex"
    "flag"
    "fmt"
    "io"
    "os"
)

func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")

```

```

flag.Parse()
key, err := hex.DecodeString(*keyHex)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
block, err := aes.NewCipher(key)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
ciphertext, err := io.ReadAll(os.Stdin)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}
iv := ciphertext[:aes.BlockSize]
plaintext := make([]byte, len(ciphertext)-aes.BlockSize)
dec := cipher.NewCBCDecrypter(block, iv)
dec.CryptBlocks(plaintext, ciphertext[aes.BlockSize:])
plaintext = Unpad(plaintext, aes.BlockSize)
os.Stdout.Write(plaintext)
}

func Unpad(data []byte, blockSize int) []byte {
n := int(data[len(data)-1])
return data[:len(data)-n]
}

```

([Listing aes/1](#))

Let's try enciphering something:

```
go run ./cmd/encipher -key \
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1 \
< tiger.txt > enciphered.bin
```

If all is well, we should see something familiar emerge from the other end of decipher:

```
go run ./cmd/decipher -key \
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1 \
< enciphered.bin
```

The tiger appears at its own pleasure...

Great. And using any other cipher in place of AES, if you wanted to, should be equally straightforward.

Encryption plus authentication

In the previous example, we used AES in CBC mode, which is fine. But we can do more. AES also provides another mode which includes authentication: Galois Counter Mode (GCM).

With AES-GCM, we get not only a strong block cipher, but a message integrity check too. It's similar to the HMAC authentication scheme we looked at in the previous chapter. Though the implementation details are different, the effect is the same: Bob can be confident that the message he received is exactly the one Alice sent.

Using AES in GCM mode is just as easy as in CBC mode, if not easier:

```
block, err := aes.NewCipher(key)
...
gcm, err := cipher.NewGCM(block)
...
nonce := make([]byte, gcm.NonceSize())
_, err = rand.Read(nonce)
...
ciphertext := gcm.Seal(nonce, nonce, plaintext, nil)
```

[\(Listing aes/2\)](#)

Just as before, we create a new cipher object with `NewCipher`. Instead of creating a CBC encrypter with it, though, we use it to create a GCM instead.

To do the actual encryption, we call the GCM’s `Seal` method. If you’re wondering why it’s called this, think about the envelope metaphor we used in the chapter on authentication. AES-GCM not only encrypts the data, it seals the ciphertext inside a tamper-proof envelope.

Enciphering with AES-GCM

With our shift encrypter, we attached the random input data (the nonce, or IV) as a prefix to the ciphertext, because the recipient needs it. AES-GCM does this too, and it also appends the *authentication tag*—the message integrity check—as a suffix.

The message that goes over the wire, then, includes these three components: the nonce, the ciphertext, and the authentication tag. Here’s how to “open”, or unseal, it at the other end:

```
block, err := aes.NewCipher(key)
...
gcm, err := cipher.NewGCM(block)
...
nonce := ciphertext[:gcm.NonceSize()]
ciphertext = ciphertext[gcm.NonceSize():]
plaintext, err := gcm.Open(nil, nonce, ciphertext, nil)
```

([Listing aes/2](#))

Another nice feature of AES-GCM is that it does its own padding, so we can skip those steps. In fact, the code above is basically all we'll ever need to write in Go to do modern cryptography. How convenient!

The final final version

Let's use GCM to put the finishing touches to our enciphering and deciphering tools, then.

GOAL: Update the encipher and decipher programs to use AES-GCM.

HINT: The only changes you need here are to call `seal` and `open` instead of `CryptBlocks`, and to extract the nonce from the first part of the enciphered message before decrypting it.

SOLUTION: Here's the GCM version of `encipher`:

```
package main

import (
```

```
"crypto/aes"
"crypto/cipher"
"crypto/rand"
"encoding/hex"
"flag"
"fmt"
"io"
"os"
)

func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    block, err := aes.NewCipher(key)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    plaintext, err := io.ReadAll(os.Stdin)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    gcm, err := cipher.NewGCM(block)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
```

```
    }

    nonce := make([]byte, gcm.NonceSize())
    _, err = rand.Read(nonce)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    ciphertext := gcm.Seal(nonce, nonce, plaintext, nil)
    os.Stdout.Write(ciphertext)
}
```

([Listing aes/2](#))

And here's decipher:

```
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "encoding/hex"
    "flag"
    "fmt"
    "io"
    "os"
)

func main() {
    keyHex := flag.String("key", "", "32-byte key in
hexadecimal")
    flag.Parse()
    key, err := hex.DecodeString(*keyHex)
```

```

if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

block, err := aes.NewCipher(key)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

ciphertext, err := io.ReadAll(os.Stdin)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

gcm, err := cipher.NewGCM(block)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

nonce := ciphertext[:gcm.NonceSize()]
ciphertext = ciphertext[gcm.NonceSize():]
plaintext, err := gcm.Open(nil, nonce, ciphertext, nil)
if err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
}

os.Stdout.Write(plaintext)
}

```

([Listing aes/2](#))

A quick double-check that everything's still in order:

```
go run ./cmd/encipher -key \
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1
\
<tiger.txt | go run ./cmd/decipher -key \
8e8c2771be5c2bb10d541a5bf6aa51203e0bce2d6d4fa267af89a6e20df11f1
```

The tiger appears at its own pleasure...

Pretty neat! So this should be everything you need to keep your secrets safe from the government, God Emperor Eve, or even your kid sister.

Weaknesses

Is AES really *that* secure, though? Well, yes. There are more sophisticated and technically more secure ciphers in existence, and more are being invented all the time, but we don't need to worry about those yet. AES, used correctly, with 256-bit keys, is secure enough for any imaginable cryptographic application today.

Tomorrow, of course, is another day, and we'll say a word about that at the end of this chapter.

Yes, there are theoretical weaknesses in AES. Cryptographers love to find theoretical weaknesses in ciphers, not surprisingly: that's literally their job. However, the kind of attack that would excite an academic researcher might not be of huge concern to you, me, Alice, or Bob.

For example, the sort of weakness we're talking about might leak one or two bits of the key, reducing the brute-force search time by a factor of two or four. But, as we've seen in previous chapters, with 256-bit keys this time is of the order of billions of trillions of years!

Just cutting that time in half isn't nothing, to be sure, but it's nothing for *us* to worry about. In other words, the known vulnerabilities in AES are minor and of purely academic interest.

Even if AES isn't quite the strongest known cipher, the fact that it's a *standard* is a big point in its favour. For one thing, lots of resources are focused on trying to analyse and break it. If a practical attack on AES is ever found, we'll know about it pretty quickly, because it will be in the headlines.

Implementation fails

To repeat, used correctly, AES is secure. "Used correctly", though, is doing a lot of work in that sentence. There are many examples of AES being used *incorrectly*, and in that case all bets are off.

For example, Bruce Schneier [reports](#) that, until fairly recently, the Zoom videoconferencing app used AES in ECB mode, and we know why *that's* no good. The US Internal Revenue Service mandated ECB mode for encryption for a while, too. Maybe someone should send them a copy of this book.

The IRS also required its users to use an IV consisting of all zeroes, which is about as secure as using the password "password". Let's hope they're better at calculating taxes than they are at cryptography.

In another case, the auto maker Hyundai secured its in-car entertainment system software using AES, but with a single global key. Unfortunately, it was an [example key](#) that had been widely used in internet encryption tutorials. (Whoops.)

Even lightbulbs aren't safe, it seems. The popular Philips Hue smart lighting system was successfully [hijacked](#) by security researchers in a way that could theoretically have turned an entire city's lightbulbs into a malicious botnet. A flaw in the implementation allowed the researchers to extract the global signing key using a side channel attack. (Lights out!)

You get the point. These are dumb mistakes, but don't blame the developers. Blame the cryptographically-illiterate managers putting those devs under intense pressure to finish and ship, security be damned. If you find yourself in a similar predicament, at least now you're armed with what you've learned in this book, and you can use it to push back.

Mashing the keys

Even when a secure cryptosystem *is* used correctly and there are no flaws in the software, things can still be sub-optimal. This AES-encrypted USB flash drive features its own built-in keypad:



To secure your data, you just tap in your key. Cool, and it's very [Mr. Robot](#).

But to use a full 256-bit key would require 77 button presses: not something even the most paranoid users are probably going to bother with. In fact, keys are limited to 10 digits anyway, and early versions made it trivial to [erase the key altogether](#) by holding down a couple of buttons.

Nice idea, shame about the implementation.

Tomorrow

Now that we've had a good laugh, let's sit up and try to look serious for a moment as we contemplate the future of cryptography.

The future is quantum

Again, most people agree that AES, used correctly, is secure *today*. That won't be true forever, of course, because new vulnerabilities will be found, and brute-force attacks will get faster. So how long does AES have? What kind of time horizon are we talking about?

You've probably heard of *quantum computers*. So what the heck are they, and what do they have to do with cryptography?

We saw in the chapter on randomness that the outcomes of certain quantum measurements, such as the polarization of a photon, are in principle unpredictable. They're fundamentally *probabilistic*, rather than deterministic.

If we label the possible measurement outcomes as “0” and “1”, for example, then we can use the quantum state of the photon to represent one bit of information. That sounds like something we could do computation with, and it is. Let's see how.

Quantum parallelism

Suppose we wanted to compute the XOR of two bits, as in the AES algorithm we examined earlier in the chapter. If we consider XOR as a function, there are four possible results, one for each possible combination of the two inputs:

x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

To calculate all these results with an old-fashioned *classical*—that is, non-quantum—computer, we'd have to call the function four times, once for each of the possibilities.

That's because classical bits can only represent one value at a time: either 0 or 1. On the other hand, quantum bits—*qubits*—can represent both values at once: a superposition of 0 *and* 1.

A computer made of qubits, then, could calculate all the possible outcomes of the XOR function in a single operation. It's a kind of parallel processing: *quantum parallelism*.

We saw in the chapter on enumeration that, while classical parallel processing does help in brute-force attacks, by trying multiple keys at once, it doesn't help *much*. That's because to try N keys at once, you need N CPUs, and that's expensive for large N .

Quantum parallelism is different. You only need one CPU (albeit a quantum one), and the parallelism comes free.

For example, suppose you want to crack a 256-bit AES key, and you had a computer with 256 qubits. Each of those qubits could stand for the corresponding bit of the key. Since it can simultaneously represent both possible values of that bit (0 and 1), you could effectively try all the possible keys at once. It doesn't work quite like that in practice, but it does still give us a huge speedup, exponential with the number of qubits.

Similarly, the security of systems like RSA and Diffie-Hellman-Merkle relies today on the difficulty of factorising and finding logarithms of very large numbers, because the only way to do it is to try lots of possibilities. But a powerful quantum computer would be *able* to try lots of possibilities at once, making this problem easy.

That sounds like great news for Eve, and equally bad news for Alice and Bob. But is it?

The catch

Well, hold your horses. First, it's awfully difficult (and expensive) to build even one qubit, never mind 256 of them.

The reason is that to avoid destroying the superposition of states in a qubit (or rather, to avoid becoming entangled with that superposition *yourself*), you mustn't measure it. Unfortunately, almost any kind of particle-level interaction counts as a "measurement": the qubit being hit by a stray photon, for example.

Second, quantum technology has some advantages for Alice and Bob, too. One of them is that it lets them exchange messages in such a way that they can't be undetectably modified in transit by Mallory, a risk that we discussed in the chapter on authentication.

In fact, they can't even be *read* by a third party without detection. Again, this is because measuring the state of a quantum bit collapses the superposition (from the observer's point of view).

The catch, though, is the same as it was for Eve: Alice and Bob need to keep their quantum bits isolated from interactions with other particles, which would cause *decoherence*: reducing the qubit to a plain old classical bit. If Mallory or Eve tries to read the state of a message qubit, they destroy the superposition, erasing the message. But the same thing happens if the qubit is hit by a stray photon, too.

Why your computer isn't quantum... yet

And there are a great many stray photons bouncing around the universe. You're being hit by billions of them at this very moment, for instance. Isolating a qubit from all interactions for long enough to finish a computation with it is a tricky business, and it takes a lot of liquid helium. This makes the quantum computers of today very large, very cold, very expensive, and so inefficient that for all practical purposes a plain old classical computer is far better.

But classical computers, too, used to be enormous, hulking machines that needed reinforced floors, and were only available to governments or those with very deep pockets. Now you can fit one *in* your pocket, and you probably do.

So we can also expect quantum computers to improve pretty rapidly, reaching a level in the foreseeable future where they can be used for practical attacks on encrypted messages. This inflection point is known by researchers as *Q-Day*: the day when current cryptosystems like AES and RSA can no longer be considered secure.

We don't know exactly when Q-Day will be. It might be decades away, or it might be tomorrow, but it's definitely coming.

What then?

Post-quantum cryptography

Well, we'll need a new kind of cryptography: one that's secure against attacks not just by classical computers, but by quantum computers too. Research into this *post-quantum cryptography* is already well under way: nothing like getting out ahead of the problem.

How is that possible, since the researchers don't yet have workable quantum computers to try out their algorithms on?

Well, even if we can't buy one off the shelf, it's perfectly possible to *imagine* how a quantum computer would work, and even to simulate one (slowly). So, while the current efforts are mostly theoretical, they're also quite sound (at least, as far as we can tell).

There are a few different approaches to strengthening cryptosystems against quantum attack. One is simply to use larger keys. When quantum computers become big enough and reliable enough to threaten the security of 256-bit keys, we can just keep adding more bits until we feel safe again. As with classical computers, it's a lot more expensive for Eve to add another qubit to her computer than it is for Alice to add another bit to her key.

More fundamentally, we can invent new kinds of cryptographic problem: ones that are as hard for quantum computers to solve as factorisation is for classical computers. Some of these, such as [lattice-based cryptography](#), and quantum-resistant [hashing](#), look promising.

Maybe post-quantum cryptography will be the last cryptography, or maybe not. There may be another computing revolution out there, waiting to be found, that's as significant a speedup as quantum computing. It seems unlikely, but you never know.

Nature never seems to run out of fresh surprises for us, so stay tuned.
Things are going to get even more interesting.

Afterword

Oh, you wanted to know what happened with Alice and Bob? Well, they exchanged keys (securely), had many enjoyable pizza dates, and, despite the attempts of Eve and Mallory to interfere and subvert their communications, are still an item to this day.

Now that Alice is studying for her doctorate in post-quantum cryptography, she's very busy, but she still has time to leave affectionate notes for Bob on the fridge: encrypted, of course.

Even 256-bit keys won't last forever, but maybe love can. We'll see.

About this book



Who wrote this?

[John Arundel](#) is a Go teacher and mentor of many years experience. He's helped literally thousands of people to learn Go, with friendly, supportive, professional mentoring, and he can help you too. Find out more:

- [Learn Go remotely with me](#)

Feedback

If you enjoyed this book, let me know! Email go@bitfieldconsulting.com with your comments. If you didn't enjoy it, or found a problem, I'd like to hear that too. All your feedback will go to improving the book.

Also, please tell your friends, or post about the book on social media. I'm not a global mega-corporation, and I don't have a publisher or a marketing budget: I write and produce these books myself, at home, in my spare time. I'm not doing this for the money: I'm doing it so that I can help bring the power of Go to as many people as possible.

That's where you can help, too. If you love Go, tell a friend about this book!

Free updates to future editions

All my books come with free updates to future editions, for life. Make sure you save the email containing your download link that you got when you made your purchase. It's your key to downloading future updates.

When I publish a new edition, you'll get an email to let you know about it (make sure you're [subscribed](#) to my mailing list). When that happens, re-visit the link in your original download email, and you'll be able to download the new version.

NOTE: The Squarespace store makes this update process confusing for some readers, and I heartily sympathise with them. Your original download link is only valid for 24 hours, and if you re-visit it after that time, you'll see what *looks* like an error page:



But, in tiny print, you'll also see a “click here” link. Click it, and a new email with a new download link will be sent to you.

Join my Go Club

I have a mailing list where you can subscribe to my latest articles, book news, and special offers. I'd be honoured if you'd like to be a member. Needless to say, I'll never share your data with anyone, and you can unsubscribe at any time.

- [Join Go Club](#)

For the Love of Go

[For the Love of Go](#) is a book introducing the Go programming language, suitable for complete beginners, as well as those with experience programming in other languages.

If you've used Go before but feel somehow you skipped something important, this book will build your confidence in the fundamentals. Take your first steps toward mastery with this fun, readable, and easy-to-follow guide.

Throughout the book we'll be working together to develop a fun and useful project in Go: an online bookstore called Happy Fun Books. You'll learn how to use Go to store data about real-world objects such as books, how to write code to manage and modify that data, and how to build useful and effective programs around it.

The [For the Love of Go: Video Course](#) also includes this book.

The Power of Go: Tools

Are you ready to unlock the power of Go, master obviousness-oriented programming, and learn the secrets of Zen mountaineering? Then you're ready for [The Power of Go: Tools](#).

It's the next step on your software engineering journey, explaining how to write simple, powerful, idiomatic, and even beautiful programs in Go.

This friendly, supportive, yet challenging book will show you how master software engineers think, and guide you through the process of designing production-ready command-line tools in Go step by step.

Further reading

You can find more of my books on Go here:

- [Go books by John Arundel](#)

You can find more Go tutorials and exercises here:

- [Go tutorials from Bitfield](#)

I have a YouTube channel where I post occasional videos on Go, and there are also some curated playlists of what I judge to be the very best Go talks and tutorials available, here:

- [Bitfield Consulting on YouTube](#)

Credits

Gopher images by [MariaLetta](#).

Acknowledgements



Many of the ideas and programs in this book were developed with the indispensable help of my students at the [Bitfield Institute of Technology](#), the online school where I teach and mentor Go, and software engineering in general. They are probably now pretty fed up with hearing me talk about cryptography.

Special thanks are due to Joumana El Alaoui, who read practically all of the book in draft form, and made many helpful suggestions. Now it's her turn to write a book, and I'm looking forward to it.