EXPLORE GO CRYPTOGRAPHY

"I laughed, I learned. It's a great book."
—Dian Amencourt

A GOPHER'S GUIDE TO CIPHERS

JOHN ARUNDEL

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 TestDecipherWithKey1TransformsIBMToHAL(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*.