Cryptography in Context

Nathaniel Karst

November 19, 2014

Preface

It is safe to say that the world as we know it could exist without modern communication systems. But from online credit card transactions to secure messages between organizations and their agents, these technologies allow us to trust each other in an increasingly digital world. In these notes, we will begin to unpack these technologies, from both mathematical and practical perspective. You will have the opportunity to implement many of these communication protocols yourself, and in so doing, hopefully gain a deeper understanding and appreciation for the underlying ideas. We will regularly confront the ethical issues that arise at the interface of these theoretic tools and their real world implementation, as the use and abuse of modern cryptography has serious implications for the trajectory of our society.

How To Use These Notes

Content areas are intended to be relatively stand-alone. Each content area discusses the overall idea behind the topic, and then proceeds to Matlab implementation. (While I have not thoroughly tested it, Octave should work as a good substitute.) Each content area has associated studio problems located in Appendix A. These problems are intended to be started in class with the help of an instructor and completed outside class. The solutions for studio problems are located in Appendix B. Teaching notes for instructors, including learning objectives and discussion questions where appropriate, are including in Appendix C.

Acknowledgements

I would like to thank the Teaching Innovation Fund at Babson College for supporting the development of these notes and exercises. My thanks also goes to the Spring 2014 cohort of Cryptology and Coding Theory for its help in molding these notes for general consumption.

Contents

Contents

Chapter 1

Introduction

Cryptography is just one of a collection of technologies that allows us to communicate with one another in a digital world. In fact, these technologies can be abstracted outside the digital context into a so-called "communication system." This theoretic object is helpful in allowing us to examine what we consider important in a communication scheme and how these various important pieces might fit together. The general communication systems also give us the opportunity to introduce some important archetypical characters, namely Alice, Bob, and Eve, into our vocabulary.

In both the classical and modern cryptosystems that we will study, you will have the chance to implement the system itself and a successful attack on that system, when one exists. This implementation is incredibly helpful, as it allows us to see the details of a particular system in action. These exercises do require some experience programming. We'll assume here that the typical student has no prior knowledge. We'll begin with some general examples and move into more cryptographic applications as we gain confidence.

1.1 Communication Systems

The hardware and software components of communications systems are knit together by deep and fascinating mathematics. In the most general framework, we consider one party, traditionally named Alice, who wants to send a secret message to another party, traditionally named Bob. Alice can only communicate to Bob through a noisy public **channel** (*e.g.*, postal service, hardwired connection, radio, Wifi) which is monitored by an eavesdropper, traditionally named Eve, who "hears" everything that Alice says to Bob. In a full communication system, there are three main processes that are completed before Alice transmits her message to Bob: **compression**, **encryption**, and **protection** as seen in Figure 1.1. This conceptual framework was formalized by Claude Shannon and Warren Weaver in the mid-1940s. Shannon went on to develop the field of **information theory**, an enormously important subject which today encompasses all compression, encryption, and protection technologies.

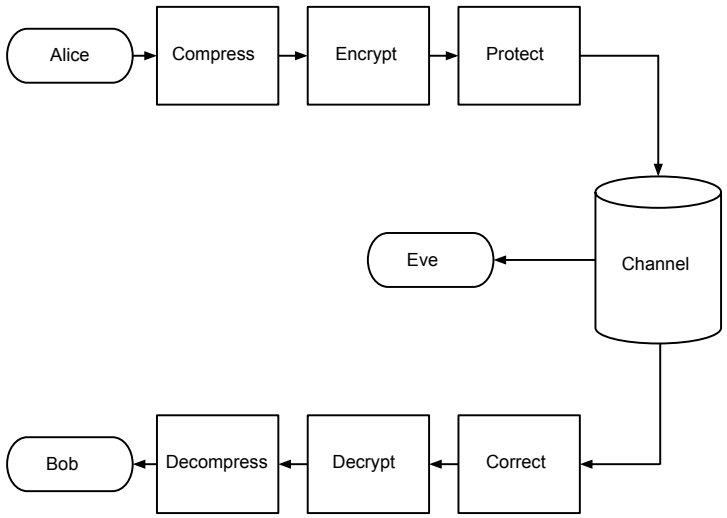

Figure 1.1: General communication system with Alice sending a message to Bob through a public channel monitored by Eve.

1.1.1 Compression

In order to save time and energy, Alice typically wants to send as short a message as possible while getting her point across. She can use a suite of mathematical techniques broadly called "compression" to make her message as small as possible while retaining most or all of its meaning. For instance, if Alice were trying to send the message "Let's meet together tonight at 9 o'clock on the Boston

Common. Looking forward to seeing you then!", a very simple compression scheme would reduce this message to "Meet tonight nine pm Boston Common." While perhaps not as polite, this new message certainly conveys the same basic idea as the original. You can imagine that compression techniques get quite a bit more complicated than this small example, but the idea is still the same: remove redundancy from the message.

Example 1.1.1. *Imagine a census agent wants to transmit the aggregate genders she counted in a particular day. The list reads*

`M, F, M, F, F, F, N, M, M, ..., F`

There are 505 females (`F`*), 490 males (*`M`*), and 5 individuals who preferred not to respond (*`N`*) in the sample. How might she compress her message?*

Rather than send 1,000 different characters, the census agent might send something like `505F, 490M, 5N`*. Notice that she could have also sent the groups in a different order without changing the message.*

1.1.2 Encryption

Since Alice wants her message to Bob to be secret, she somehow needs to make the message she sends across the public channel unintelligible to everyone but Bob. She can accomplish this with a number of mathematical techniques broadly termed "encryption." An encryption technique matches each character in the original message (called the **plaintext** in encryption protocols) into a unique character in the encrypted message (called the **ciphertext** in encryption protocols). As far as displaying these messages, the traditional approach is to

- Use only capital letters
- Ignore punctuation and other non-alphabetic characters
- Group characters in small sets in order to minimize human errors in processing

We will ask ourselves how our cryptographic tools would have to change if we ignored some or all of these bullet points, but for now, let's stick to the easier case.

Example 1.1.2. *Suppose Alice and Bob could have agreed that an* `A` *in plaintext will become a* `B` *in the ciphertext. Similarly, a* `B` *would turn into a* `C`*, and so on. Using the display conventions laid out above, Alice's compressed message "Meet tonight nine pm Boston Common" would read*

`MEETT ONIGH TNINE PMBOS TONCO MMON`

Using the encryption scheme outlined above, Alice's message would be transformed in the following way:

`NFFUU POJHI UOJOF QNCPT UPODP NNPO`

While this ciphertext looks like complete gibberish, both Alice and Bob know that a meaningful message lies underneath. But what about our eavesdropper Eve? We could assume that she knows nothing about how Alice and Bob have decided to encrypt their messages. This approach is called **secrecy through obscurity**; Alice and Bob keep everything about their encryption scheme secret, and in that way keep Eve from deciphering their messages. The issue here is that if Eve somehow manages to get an idea of how the message is being encoded, then Alice and Bob will *underestimate* her capabilities, which is in general a very bad thing to do to an adversary. Instead we typically follow **Shannon's maxim** (also called **Kerckhoff's law**) which says in short "The enemy knows the system." In more detail, we assume that Eve knows everything about how the message was encrypted *except* for a special piece of secret information call the **key**. In Example 1.1.2 for instance, we assume that Eve knows that Alice and Bob are encrypting by substituting one letter for another, but that she does not know the secret substitution rule. If Eve wants to break the code, she needs to figure out, in one way or another, what the substitution rule is. Both the encryption process, called **cryptography**, and the breaking of ciphers, called **cryptanalysis**, are huge fields, each with their own beautiful and powerful results. We will discuss both in the context of several cryptosystems in these notes.

1.1.3 Protection

The final step Alice performs before transmission is protection. The public channel across which Alice and Bob communicate is a noisy place. We assume that Alice and Bob are not the only pair of people trying to communicate over this space. Moreover, many channels have inherent background noise that makes hearing and understanding a message difficult. For a physical example, imagine Alice is trying to tell Bob a secret in a crowded restaurant. Other pairs of people are talking, there's noise from the kitchen, and the music playing is drowning out much of the conversation. What can Alice do to get her message across? She could talk louder. Or perhaps she could repeat herself several times, in the hopes that Bob could piece together her meaning. We'll see that these are only two of the most basic attempts at message protection. There are sophisticated mathematical techniques that allow Alice to systematically introduce extra information into her message so that Bob will be able to reconstruct her meaning, even if some of her message is corrupted during transmission.

1.1.4 Representations of Data

In the classical cryptosystems discussed in Chapter 2, messages are traditionally represented in uppercase letters without any spaces, punctuation, or other non-alphabetic characters. In many cryptosystems, however, it is convenient to be able to perform arithmetic operations when turning plaintext into ciphertext. This necessitates a consistent method for converting numbers to letters. The

most straightforward is to associate the 26 letters in order with the numbers 0, 1, . . . , 25:

$$\texttt{A} \leftrightarrow 0,\ \texttt{B} \leftrightarrow 1,\ \texttt{C} \leftrightarrow 2, \ldots, \texttt{Z} \leftrightarrow 25.$$

It may seem more natural to begin number at 1 instead of 0, but we'll see over the course of our investigations that the presence of zero is extremely useful, and indeed necessary, in many cryptosystems. We can therefore think of strings of characters and arrays of numbers interchangeably and will frequently do so without further mention. Rectangular arrays of numbers with either one row or one column are called **vectors**. A rectangular array with r rows and c columns is called an $r \times c$ **matrix**. Vectors are a subset of matrices in the same way that squares are a subset of rectangles. We will often think of messages in terms of both vectors and matrices.

Example 1.1.3. *The following array of characters and the numerical vector are equivalent:*

$$\texttt{ZEBRA} \leftrightarrow [25 \quad 4 \quad 1 \quad 17 \quad 0].$$

Example 1.1.4. *The following array of characters and the numerical matrix are equivalent:*

$$\begin{matrix}\texttt{L} & \texttt{O} \\ \texttt{V} & \texttt{E}\end{matrix} \leftrightarrow \begin{bmatrix}11 & 14 \\ 21 & 4\end{bmatrix}.$$

In the modern cryptosystems discussed in Chapter 3, messages are traditionally represented by strings of 0s and 1s. Each 0 or 1 is called a **bit** (a contraction of the phrase "binary digit"). We can represent text as a bit string by representing each character as itself a bit string and then concatenating, that is appending one after the other. Perhaps the most popular way to accomplish this conversion is the ASCII (American Standard Code for Information Interchange). The ASCII "alphabet" is extensive. For instance, the ASCII bit string for the letter `A` is 1000001, the ASCII bit string for the letter `a` is 1100001, and the bit string for the character `!` is 0100001. For a full list, check out the ASCII Wikipedia page (click here).

The good news is that we rarely have to deal with this conversion explicitly. Almost always it is enough to know that we can convert any string of characters into a unique string of bits, and vice versa. We can therefore think of encoding just the bits themselves, and leave the worrying about the conversion to a computer. We'll discuss this paradigm in much more detail at the beginning of Chapter 3.

1.2 Matlab

Matlab is an industry-standard computational package developed by MathWorks. In this section, we will cover some core computer programming ideas and their corresponding implementation in Matlab.

1.2.1 Basic Arithmetic

At its most basic, Matlab is a high powered calculator. For instance, if we enter `1 + 2` in the **command line**, we observe

```
>> 1 + 2

ans =

     3
```

Sometimes we don't want to show the output of a particular computation. We can **suppress** the output by appending a semicolon:

```
>> 1 + 2;
>>
```

Note that Matlab has still performed the computation; it just hasn't shown us the results. Basic arithmetic works much the way you'd probably expect, with +, `-`, `/`, `*`, and `^` all performing their traditional roles.

1.2.2 Variable Assignment

We often need to use the result of a computation for some other purpose. We can store the results of a computation using **variable assignment**. An assignment always has the form

`variable name = variable value`.

We assign the value on the right of the equals sign to the variable name on the left of the equals sign. For example,

```
>> x = 1 + 2;
>> x

x =

     3
```

The variable `x` now has value 3 and will continue to have this value until we overwrite it or clear it.

Variable assignment can produce expressions that might not make sense at first look. For instance, consider

```
>> x = 1 + 2;
>> x = 2*x;
```

The last line seems strange from a mathematical perspective. If we consider this to be an equation, there is only one value of `x` that satisfies the constraint. But this is *not* an equation! It's a variable assignment! So we take the value on the right side of the equals sign and assign it to the variable name on the left side of the equals sign. After the first line executes, we have `x = 3`. The value on the right side of the equals sign on the second line is therefore 6. So the second line actually overwrites the old value of `x` with a new value of 6. While these types of assignments are perfectly legal, and in some cases desirable, they can also easily confuse people who are trying to read your code. Be careful using them.

1.2.3 Arrays

One of the strongest aspects of Matlab is that it can deal with arrays and matrices very efficiently. We can generate arrays very simply.

```
>> a = 1:5

a =

     1     2     3     4     5
```

Notice that we could've arrived at the same result by entering

```
>> a = [1, 2, 3, 4, 5];
```

We can see that `a` is a 1×5 matrix, also called a **row vector**.

```
>> size(a)

ans =

     1     5
```

To see exactly what the `size` function is telling us, we can type "`help size`" in the command line.

We could turn `a` into a **column vector** `b` by appending a single apostrophe.

```
>> b = a'

b =

     1
     2
     3
     4
     5
```

This operation is called the **matrix transpose**, and it serves to turn the rows of one matrix into the columns of another, and vice versa. Notice that we could've arrived at the same result by entering

```
>> b = [1; 2; 3; 4; 5];
```

Here, the `size` function gives

```
>> size(b)

ans =

     5     1
```

Notice that now our vector has $r = 5$ rows and $c = 1$ column. Using the character ";" when defining a matrix or vector tells Matlab to start a new row.

The `size` function is similar to the `numel` function.

```
>> numel(a)

ans =

     5
```

Here Matlab returns the total number of elements located in a vector or matrix.

Matlab gives us easy way to access just part of the vector `a`. Matlab begins indexing the first element in any array as 1. We could access elements 1 through 3 (line 1 below), elements 2 to 4 (line 2 below), or elements 2 all the way to the end of the vector (line 3 below), or just the fourth element (line 4 below).

```
>> a(1:3);
>> a(2:4);
>> a(2:end);
>> a(4);
```

This type of operation is known as **slicing** the vector. Matlab also allows us to perform arithmetic on arrays. For instance,

```
>> 2*a;
>> a+5;
```

both perform the computations you'd expect. We need to be a little more careful with multiplication, however.

```
>> a*a
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

As we'll see, matrix and vector multiplication have their own definitions; unlike real or complex numbers, not any two vectors can be multiplied together. Notice that multiplying an vector by a number (also called a **scalar**) was no problem; it's only matrix-matrix operations that can get tricky.

1.2.4 Scripts and Functions

In Matlab, there's no way to save your work if you're working from the command line. The good news is we can package a series of commands into a file called a **script**, save it, and run it at any time in the future. To open a blank file, either go to `File > New > M-File` or press `CTRL+N`. The commands we enter into the script are exactly the same as if we were entering them into the command line.

For more complex tasks, we need multiple pieces of code that work together. If we put these code chunks all together, things could get confusing. Instead, we separate out blocks of code that perform specialized operations into **functions**. The Matlab syntax that defines a function is

```
function return_variable = function_name(argument1, argument2, ..., argumentN)
     operations
     ...
     return_variable = some_expression
end
```

The function `function_name` will accept the inputs, also known as **arguments**, and **return** the value of the variable `return_variable`.

For an example, imagine we wanted to define a function that takes two inputs x and y, and produces a number z according to $z = x^2 + y^2 + 2xy$. Our function might look like

```
function z = ourFirstFunction(x,y)
    z = x^2 + y^2 + 2*x*y;
end
```

For our purposes, we will always have just one function per file. The file name and the function name should match.

1.2.5 Loops

Suppose we wanted to print the numbers from 1 to 100. We could enter one command for each of the 100 numbers, but this seems repetitive. This type of repetition comes up a lot in programming, and computer scientists have given us structures called **loops** to help with automating these tasks. The simplest is the `for` loop. For instance, the for loop that would print the numbers 1 to 100 reads

```
for i = 1:100
    i
end
```

In words, this loops reads, "For `i` equals 1 to 100, perform the operations in the middle of the loop." Here the operation is just to print `i`, but you can imagine plenty of more complex scenarios.

Matlab's designers have also given us ways to perform different operations in different circumstances. Consider the following extension of our `for` loop.

```
for i = 1:100
    if i < 25
        'small'
    elseif i >= 25 & i < 75
        'medium'
    else
        'large'
    end
end
```

Let's examine this loop piece by piece. First, we see that the loop will increment `i` from 1 to 100, executing all the commands in the middle before moving on to the next value of `i`. Inside the loop, we see an **`if` statement**, which allows us to choose a single block of code to execute depending on conditions that we set. The command below the `if` is executed if and only if the `if` statement is true. For instance, here the word `small` will print if and only if `i` is less than 25. If this happens, the *whole* `if` statement is complete, and the `for` loop proceeds on to the next value of `i`. If it does not happen, then we continue to the `elseif` statement.

If we reach the `elseif` statement, we will print the word `medium` if and only if `i` is greater than or equal to 25 *and* less than 75. If this happens, the whole `if` statement is complete, and the `for` loop moves on to the next value of `i`. If we do not print `medium`, we move on to the `else` statement. If we reach the `else` statement in an `if` statement, we automatically execute the code between `else` and `end`. We can think of `else` as meaning, "if everything else has failed, do the following..."

Loops and conditional statements alone allow us to develop powerful techniques for solving complex problems. Other computer science topics will be introduced as needed throughout the text.

1.2.6 Provided Functions

There are several specialized Matlab functions bundled with these notes. Below is a short description of each. In order to learn more about one of these functions (or any other function) in Matlab, type `help` followed by the function name to get a short description.

- `frequency`: computes the relative frequency of upper-case characters in a string
- `friedman`: performs Friedman analysis for polyalphabetic cryptanalysis
- `lettersToNumbers`: converts uppercase letters to numeric values; works only on strings of characters
- `loadText`: loads text from and convert to a numeric vector
- `modInv`: computes the multiplicative inverse of x modulo n if one exists

- `numbersToLetters`: converts numeric values to uppercase letters; works for both vectors and matrices
- `ord`: computes the multiplicative order of x in $\mathbb{Z}_n$
- `powMod`: computes the $x^y \bmod n$ in a way that can handle large values of x and y
- `preprocessText`: removes all non-alphabetic characters from the input text, and convert all letters to uppercase
- `sha0`: a truncated version of the defunct SHA1 cryptographic hash; returns a 16 bit digest
- `totient`: computes Euler's totient function $\phi(n)$

Chapter 1. Introduction

Chapter 2

Classical Cryptosystems

Classical cryptosystems are designed to disguise plaintext composed of alphanumeric characters. These technologies have been employed in one form or another for much of recorded history. As cryptanalytic technology has evolved, so too have encryption methods. At this point, there are dozens of different variations of the common themes of transposition, forming a ciphertext by rearranging the letters in the plaintext, and substitution, forming a ciphertext by substituting one letter for another. We will investigate a sample of these methods, highlighting the strengths and weaknesses of each. This discussion will lead us naturally to the modern cryptosystems discussed in Chapter 3.

2.1 Transposition Ciphers

Transposition ciphers were modern technology nearly over 2700 years ago, when the Greeks used specialized rods called **scytales** (which rhymes with "Italy-s") for quick enciphering and deciphering. Alice and Bob would begin with identical syctales. Alice would wind a strip of writing material around her scytale and begin writing her message left to right along the rod. An example of this process using the plaintext "Cryptology is awesome" can be seen in Figure 2.1.

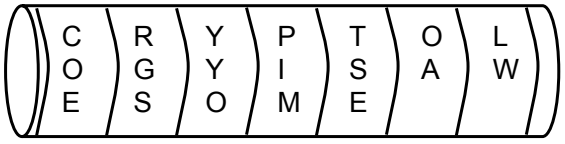

Figure 2.1: Example of transposition encoding on a scytale. Typically the message would wind all the way around the scytale; forced perspective is used here for clarity.

The plaintext makes perfect sense when it is wound around the scytale. When Alice unwinds the leather strip, however, the text is suddenly unintelligible as seen in Figure 2.2.

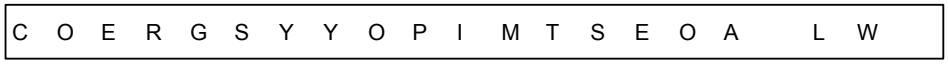

Figure 2.2: Leather message strip from Figure 2.1 unwound from the scytale. Blank spaces towards the end of the strip correspond to the blank spaces at the bottom right of Figure 2.1.

To decode the ciphertext, Bob need only wrap the leather strip around his identical scytale and read the letters off left-to-right. Notice if the two scytales were different sizes, the letters would not line up correctly.

2.1.1 Scytales to Matrices

We can bring this sort of encoding into a more modern framework. Alice and Bob first agree on a number k, called the key, which serves the same role as the scytale: a shared secret between Alice and Bob. Alice then writes her plaintext in a rectangular array with k rows. If n is the length of the plaintext, this implies there are $c = \lceil n/k \rceil$ (read "ceiling of n/k", and mathematically defined as n/k rounded up) columns in the matrix. For instance, Alice's plaintext "Cryptology is awesome" has length $n = 19$. And if, for instance, Alice and Bob agree on the key $k = 3$, then Alice's rectangular array will have $c = \lceil 19/3 \rceil = 7$ columns:

```
CRYPTOL
OGYISAW
ESOME
```

Notice that the relative placement of all the characters are the same as if Alice had written her message using a scytale as in Figure 2.1. Alice then constructs the ciphertext by reading off the columns of the matrix. There's one detail that needs to be addressed: what to do with the empty positions in the bottom right corner of the rectangular array. Here, Alice needs to *pad* her message with extra characters in order to fill out the array. Any characters will do, and Alice will trust that Bob will be able to recognize that the final characters is unnecessary once he reconstructs the plaintext. If we were to pad the bottom right-hand positions with `Z`, the ciphertext would read

```
COERG SYYOP IMTSE OAZLW Z
```

Notice that this is identical to the ciphertext in Figure 2.2, expect that the blank spaces in Figure 2.2 are occupied by Alice's padding character `Z`.

In order to decipher the ciphertext Alice has given him, Bob first constructs his own blank rectangular array. He knows the secret number k, which dictates the number of rows in the matrix. He can also knows that he must have $c = n/k$ columns, where here n is the length of the ciphertext. (Notice the ceiling function is gone here, because Alice's padding has insured that the ciphertext length n is divisible by k.) Bob then reads off the rows of his array to recover the plaintext.

Let's think about what the eavesdropper Eve can get out of this transaction. We assume that Eve has somehow intercepted the ciphertext. But Eve does not know the secret key k. In terms of the deciphering process, Eve does not know how many rows she should have in her matrix. Her only resort is to guess-and-check using different keys. Performing this process by hand could take quite a while, but a computer can easily chew through thousands of possible keys in a very short amount of time.

2.1.2 Encryption in Matlab

Let's say we have our message in an array of characters. For instance,

```
plaintext = 'CRYPTOLOGY IS AWESOME'
```

Let's assume that Alice and Bob have decided on key $k = 3$. Alice first pads her message so the total length is (evenly) divisible by 3.

```
plaintext = 'CRYPTOLOGY IS AWESOMEZZ'
```

Alice wants to turn her vector into a matrix with 3 columns and 7 rows with the text read along the rows. Matlab has a command for this sort of thing: `reshape`. To learn how it works, we type `help reshape` in the command line:

```
>> help reshape
 RESHAPE Change size.
    RESHAPE(X,M,N) returns the M-by-N matrix whose elements
    are taken columnwise from X.  An error results if X does
    not have M*N elements.
```

So, `reshape` will take a matrix `X` and produce a new matrix with the same entries, only in a different shape. Seeing an example might help us get a better idea of how this would work.

```
>> X = 1:12

X =

     1     2     3     4     5     6     7     8     9    10    11    12

>> Y = reshape(X,6,2)

Y =

     1     7
     2     8
     3     9
     4    10
     5    11
     6    12
```

So reshape took our vector `X`, and produced a matrix `Y` with 6 rows and 2 columns. Also note that `reshape` listed the elements of `X` down the *columns* of `Y`. Try `reshape(1:12,2,6)`, `reshape(1:12,4,3)` and `reshape(1:12,12,1)`. Before you run each command, try to describe what you think you will see. If what Matlab actually produces differs from your expectation, try to clarify what went wrong with your prediction.

How could Alice use this to construct her character matrix? As a start she might try

```
>> plaintext = 'CRYPTOLOGY IS AWESOMEZZ';
>> plaintext = lettersToNumbers(plaintext);
>> newMatrix = reshape(plaintext,3,7);
```

Alice could view how this transformation changed her plaintext by using the `numbersToLetters` function provided with these notes:

```
>> numbersToLetters(newMatrix)
CPLYASE
RTOIWOZ
YOGSEMZ
```

This isn't quite what Alice wanted. Notice that the message is listed down the *columns* of the matrix, rather than rows as Alice intended, just as we've come to expect from `reshape`. Imagine Alice tried the following:

```
>> plaintext = 'CRYPTOLOGY IS AWESOMEZZ';
>> plaintext = lettersToNumbers(plaintext);
>> newMatrix = reshape(plaintext,7,3);
```

Then upon inspecting `newMatrix`, she would see

```
>> numbersToLetters(newMatrix)
COE
RGS
YYO
PIM
TSE
OAZ
LWZ
```

This is nearly what Alice wants, except that the columns in the matrix she has are the rows in the matrix she wants. This colums-to-rows flip is a very common operation called the **matrix transpose**, and Matlab gives us a very quick shortcut:

```
>> numbersToLetters(newMatrix')
CRYPTOL
OGYISAW
ESOMEZZ
```

By placing a single quote `'` after the matrix, Alice has told Matlab that she wants the matrix "flipped" so that its columns become the rows of a new matrix.

In order to complete the encryption, Alice needs to read the characters column-by-column. The `reshape` function can help us here, too.

```
>> ciphertext = reshape(newMatrix',1,21);
>> numbersToLetters(ciphertext)
COERG SYYOP IMTSE OAZLW Z
```

Notice that this ciphertext matches perfectly the ciphertext Alice produced by hand earlier in this section.

Putting this all together, Alice has a good start at a piece of code that performs transpositional encryption:

```
>> plaintext = 'CRYPTOLOGY IS AWESOMEZZ';
>> plaintext = lettersToNumbers(plaintext);
>> newMatrix = reshape(plaintext,7,3);
>> ciphertext = reshape(newMatrix',1,21)
```

Alice can generalize this idea into a **function**. A function is a series of commands that execute sequentially. Functions sometimes return a value or values after they finish running. Sometimes they simply perform a computation and display the result. Let's imagine that Alice wants her encryption function to rerun the cipher text. Her Matlab encoder might look something like this:

```
function ciphertext = transpositionEncrypter(plaintext, key)
    % Produced the numeric vector output CIPHERTEXT
    % from the numeric vector input PLAINTEXT using input KEY
    plaintext   = padPlaintext(plaintext, key);
    tempMatrix  = reshape(plaintext,length(plaintext)/key,key)';
    ciphertext  = reshape(tempMatrix,1,length(plaintext));
end
```

Here, the function's name is `transpositionEncrypter`. The function takes two inputs called **arguments**, namely `plaintext` and `key`. The role of each of these arguments is clear from its name; in general this is a good practice to follow. The second and third lines are **comments**, a collection of text not executed by Matlab. Comments always start with the percent symbol and are used to explain what a function, a line, or part of a line is doing, so that others can more easily understand how your code works. Here, the comment is explaining what the function is expecting in terms of inputs and what the user can expect to get as an output. Notice in particular that the comment specifies that the input `plaintext` and the output `ciphertext` must be numeric vectors, not strings of characters.

The first thing the function does is run (also called "invoke" or "call") another function `padPlaintext` that is located in the same directory. The purpose of `padPlaintext`, like the function `transpositionEncrypter`, is clear from its name. This is a good practice to develop, as it makes it easier to share your code, or even remember what your own code is doing if you come back to it several months, weeks, or years from now. The remainder of the function is much the same as what we wrote for Alice above, except that instead of using $k = 3$, we now use the input `key`, and instead of using $n = 21$, we now use `length(plaintext)`. As practice, use `help length` in the Matlab command line to learn about this built-in function.

When the function reaches `end`, it is finished. We see in the first line of the function `ciphertext = ...`. This indicates that the function will return whatever the value of `ciphertext` is when the function terminates. For instance, imagine we run

```
>> plaintext = lettersToNumbers('Cryptology is awesome');
>> ciphertext = transpositionEncrypter(plaintext, 3);
```

We now have a new variable `ciphertext` in our workspace. We can see all the variables in our workspace using the command `whos` in Matlab.

```
>> whos
  Name            Size            Bytes  Class     Attributes
```

```
  ciphertext      1x21              168  double
  plaintext       1x19              152  double
```

If we removed `ciphertext =` from the first line of the function, the encryptor would return no value, and we would not have the variable `ciphertext` in our workspace. (Try this out to convince yourself.)

2.1.3 Decryption in Matlab

Decryption code will be left as an exercise to the reader with the following hint: use `reshape` to undo what Alice did, remembering that a matrix transpose may be necessary to flip the matrix into the correct orientation.

2.1.4 Cryptanalysis

The biggest clue that transposition encryption has been performed is that the **frequency** of characters does not change between the plaintext and the ciphertext; since transposition encryption simply rearranges the characters in the plaintext in order to form the ciphertext, the percentage of characters that are `A` in the plaintext is the same as the percentage of characters that are `A` in the ciphertext, and this rule holds for any other character in our alphabet.

Each language has its own character frequency. For instance, the 10 most common English characters together with their relative frequencies are found in Table 2.1. These relative frequencies vary from language to language, but the idea is always the same: some characters are much more common than others.

Character	Relative Frequency
e	12.702%
t	9.056%
a	8.167%
o	7.507%
i	6.966%
s	6.327%
h	6.094%
r	5.987%
d	4.253%
l	4.025%

Table 2.1: Relative frequency of English characters. Large amounts of English plaintext will roughly adhere to these relative frequencies.

If Eve intercepts a large piece of ciphertext and the relative frequencies of characters in the ciphertext roughly adhere to the relative frequencies found in

plaintext English, then Eve can be fairly certain that some form of transposition encryption has been performed. We will see in later sections that most encryption protocols actually change the frequency of characters between plaintext and ciphertext.

If the simple transposition encryption that we've discussed before has been performed, Eve knows that the secret key shared by Alice and Bob is simply a number. Moreover, she knows that the key must be a divisor of the ciphertext's length. We call the collection of all possible keys the **key space**. The larger the key space, the harder it is for Eve to stumble onto the correct key.

Here Eve has a relatively small key space. Given modern computational technology, she can simply ask a computer to decrypt the ciphertext using each of these possible keys. If the resulting text contains a large amount of clear English, then Eve can be relatively sure that she has arrived at the correct key. After all, the chances that an incorrect key would produce legible text are very small, especially if the ciphertext is relatively large.

We can write Eve's strategy in **pseudocode**, a sort of shorthand for a program Eve would write. There are several advantages to writing pseudocode. First, it gives Eve an intuitive framework from which she can construct an actual program. Second, it clearly conveys the ideas behind Eve's algorithm so that others can understand, even if they don't know the specifics of the programming language that Eve will end up using.

A pseudocode implementation of Eve's cryptanalytic strategy is as follows:

```
for keyGuess from 1 to length(ciphertext)
     if length(ciphertext) is not divisible by keyGuess
          continue
     end
     plaintextGuess = transpositionDecrypt(ciphertext, keyGuess)
     print plaintextGuess
end
```

This pseudocode would certainly not run in Matlab (or any other language, for that matter). But the ideas behind what Eve is trying to do should be clear. Converting this pseudocode into an actual Matlab function is left as a studio problem for this section.

2.2 Caesar Ciphers

In contrast to transpositional ciphers which simply rearrange the order of plaintext characters, substitutional ciphers replace each character in the plaintext with another character according to an agreed upon formula. The simplest family of substitutional ciphers are the **Caesar ciphers**. True to their name, this collection of techniques was deployed by Julius Caesar over 2000 years ago. In this scheme, Alice and Bob first agree on a natural number k to serve as the key. When Alice operates on the plaintext, she replaces each letter with the k^{th} letter after it. For instance, $k = 1$, then `A` becomes `B`, `B` becomes `C`, and so on, with the expectation that `Z` becomes `A`. Similarly, if $k = 2$, then `A` becomes `C`, `B` becomes `D`, and so on, until `Y` becomes `A`, and `Z` becomes `B`. Notice we exclude $k = 0$, because the ciphertext would be identical to the plaintext in this case. To decrypt, Bob simply replaces each letter in the ciphertext with the letter k positions before it in the alphabet.

For example, imagine that Alice wants to encrypt the plaintext "Meet tonight nine pm Boston Common." Then the plaintext reads

```
MEETT ONIGH TNINE PMBOS TONCO MMON
```

Using the Caesar encryption with key $k = 1$, Alice would produce the ciphertext

```
NFFUU POJHI UOJOF QNCPT UPODP NNPO
```

Notice that unlike transpositional ciphers, substitutional ciphers like the Caesar ciphers change the relative frequency of characters in the ciphertext.

2.2.1 Caesar Encryption and Modular Arithmetic

If we consider Alice numerically converting her plaintext into numerical vector form via the mapping `A` $\leftrightarrow 0$, `B` $\leftrightarrow 1$, ..., `Z` $\leftrightarrow 25$, we can think of the Caesar encryption of plaintext vector $\mathbf{p}$ into ciphertext vector $\mathbf{c}$ with key k as

$$\mathbf{c} = \mathbf{p} + k.$$

But as it stands, there is no indication that when we get to the end of the alphabet, the encryption wraps back around to the beginning. For this, we need a new mathematical concept.

You may remember learning about remainders back in elementary school when building up to the idea of long division. We saw that the remainder of 3, 7, 11, 15, ... after dividing by 4 are all 3, because after taking out a whole number of 4s from the quotient, we were left with 3. This idea comes up all the time in mathematics, engineering, and computer science. In these contexts, we use different terminology, but the meaning remains the same. We say that an integer a is **congruent** (or **equivalent**) to b **modulo** a natural number n if a and b have the same remainder when divided by n. In symbols, we write $a \equiv b \bmod n$. For some concrete examples, we have

$$7 \equiv 3 \bmod 4, \quad 162 \equiv 2 \bmod 5, \quad 34 \equiv 0 \bmod 17, \quad 3 \equiv 1 \bmod 2.$$

Note that it does not matter which side of the equation the word mod occurs; the *entire* line is a statement in modular arithmetic.

We could verify these examples using the `mod` command in Matlab. Here, $a \equiv b \bmod n$ corresponds to the command `mod(a,n)` producing the value b. For instance, the first example above would read

```
>> mod(7,3)

ans =

     1
```

Another way to look at this is to conceptualize modular arithmetic as a wrapping. Imagine we're thinking about number modulo 4. As we start counting up from 0, we have the expected 0, 1, 2, 3. When we get to 4 however, we wrap back to 0, as there is no remainder when 4 is divided by itself. Similarly, 5 now becomes 1, 6 becomes 2, and so on. In fact, every number is congruent to either 0, 1, 2, or 3 modulo 4, and so we can restrict our attention to just these numbers. We define this set symbolically as $\mathbb{Z}_4$, read "the integers modulo 4." This concept generalizes to $\mathbb{Z}_n$, the integers modulo n.

Imagine we're performing Caesar encryption with key $k = 1$ to the letter `Z` $\leftrightarrow 25$. We shift one letter forward by adding $k = 1$ to 25 to form 26. But the desired result of the encryption is `A` $\leftrightarrow 0$. So we want $26 \equiv 0 \bmod n$ for some n. The only solution here is $n = 26$, because we don't want numbers to wrap before we get to `Z`. So the correct equation describing Caesar encryption with key k is

$$\mathbf{c} = (\mathbf{p} + k) \bmod 26,$$

where here we're assuming that modular operation is applied element-wise in a vector, so that every entry in $\mathbf{c}$ is an element of $\mathbb{Z}_{26}$. In fact, this is the way modular arithmetic is done in Matlab, as we could verify either from the documentation in `help mod` or by inspection

```
>> mod(1:8,4)

ans =

     1     2     3     0     1     2     3     0
```

So Alice's encryption function might look something like this:

```
function c = caesarEncrypter(p, k)
    % affineEncrypter(p, k, ell) performs Caesar encryption to produce
    % numeric ciphertext c from numeric plaintext vector p and key k
    % according to the equation c = (p + k) mod 26.
    c = mod(p + k, 26);
end
```

2.2.2 Caesar Decryption and Additive Inverses

In general, Bob inverts Alice's operations in order to recover the plaintext from the ciphertext. Here, your intuition may be telling you that this means that Bob should *subtract* k from each element in the ciphertext vector, giving

$$\mathbf{p} = (\mathbf{c} - k) \bmod 26.$$

Your intuition is exactly right, but we haven't thought about how *negative* numbers behave in modular arithmetic. For instance, if Bob observes `A` $\leftrightarrow 0$ in the cipher text generated with key $k = 1$, he knows that he must shift the corresponding character back 1 to recover a `Z` $\leftrightarrow 25$ in the plaintext. In other words, we're observing the congruence $-1 \equiv 25 \bmod 26$.

If we go way back to the definition of negative numbers (more technically termed **additive inverses**) as we typically think of them, the quantity $-x$ is defined as the number we need to *add* to x in order to get 0 as the result. The same is true in modular arithmetic. The quantity -1 is defined as the number that we need to *add* to 1 in order to get 0 modulo 26. This sheds light on the congruence $-1 \equiv 25 \bmod 26$ that we intuitively derived in the previous paragraph. Notice that similar logic implies that every element x of $\mathbb{Z}_{26}$ has a *unique* additive inverse that is also in $\mathbb{Z}_{26}$, namely $26 - x$. This is good news, as it means that Bob can always invert the operations performed by Alice in order to recover the plaintext. What's more, these arguments are not at all dependent on the fact that we're performing all arithmetic modulo 26; we could easily perform the same operations with the same results using an alphabet with n symbols for any $n > 1$.

Matlab understands modular equivalence of negative numbers in exactly the want we've outlined above. For instance,

```
>> mod(-4:4,4)

ans =

     0     1     2     3     0     1     2     3     0
```

2.2.3 Groups

We've noticed that $\mathbb{Z}_n$ has some nice structure and properties, including a special number 0 called the **additive identity** such that $x + 0 \equiv 0 \bmod n$ for every x in $\mathbb{Z}_n$, and a unique element $-x$ for each element x called the **additive inverse** of x such that $x + (-x) \equiv 0 \bmod n$. But the goodness doesn't end there. It's also true that for elements a, b and c in $\mathbb{Z}_n$, the equation $a + (b + c) = (a + b) + c$ holds. This property is called **associativity**; notice how b "associates" with c on the left side and with a on the right side. Finally, for any two elements a and b in $\mathbb{Z}_n$, we know that $a + b \bmod n$ is an element of $\mathbb{Z}_n$. Here, we say that $\mathbb{Z}_n$ is **closed under addition**. These four properties

- Identity

- Invertibility
- Associativity
- Closure

are the defining characteristics of an algebraic structure called a **group**. If in addition, we have the propery that $a + b = b + a$ for any two elements a and b in the group, we say that the group is **commutative**; notice that the elements move around or "commute" in the equation. Man, but not all, of the groups we'll see in the course of our exploration will be commutative.

More formally, a group is a set together with an operation that acts on two elements at a time which has the four properties listed above under the given operation. For example, we have seen that the set $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ is a group under the operation of addition. Groups appear throughout mathematics and throughout cryptography, as well. We'll see the power of these simple structures in many different applications. Moreover, we'll use groups as the foundation for more interesting and complex algebraic structures that open even more applications for us.

2.2.4 Caesar Cryptanalysis

Under Shannon's maxim, we assume that Eve knows that Alice and Bob have been passing messages using a Caesar cipher and that Eve does not know the key k. Therefore, Eve knows that every letter in the plaintext has been shifted by k positions to form the ciphertext, but she doesn't know the exact value of k. In any cryptanalytic attack, Eve's task is to somehow search the key space for the correct key. Typically, she can do this in any number of ways, and over the course of these notes, we'll see several different methods.

To attack Caesar ciphers, Eve can use a technique called **frequency analysis** to try to find k. Each language has its own character frequency. For instance, the 10 most common English characters together with their relative frequencies are found in Table 2.1. These relative frequencies vary from language to language, but the idea is always the same: some characters are much more common than others.

For example, Figure 2.3 shows the relatively frequency distribution of a long novel written in English. Imagine we encrypt this novel using a Caesar cipher with key $k = 10$. The relative frequency distribution of the resulting ciphertext can be seen in Figure 2.4. Notice that since a Caesar cipher simply shifts every letter in the plaintext alphabet by k positions, the frequency distribution of the ciphertext is a shifted version of the frequency distribution of the plaintext. More precisely, the plaintext frequency distribution has been shifted by $k = 10$ positions to the right to form the frequency distribution of the ciphertext. Notice that this includes wrapping under modular arithmetic. For instance, the spike at $\mathtt{T} \leftrightarrow 19$ in the plaintext becomes a spike at $19 + 10 = 29 \equiv 3 \bmod 26$ in the ciphertext.

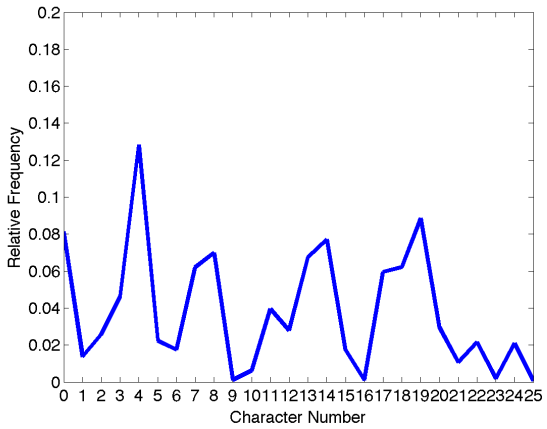

Figure 2.3: Relative frequency distribution for a long English novel. Notice that the relative frequencies correspond closely with the values listed in Table 2.1. In particular E $\leftrightarrow$ 4 is the most common character.

But how can Eve use this information in her attack? We assume that she receives all ciphertext passing through the public channel. Therefore, Eve can construct a relative frequency distribution as featured in Figure 2.4. If she assumes that the biggest spike in the frequency distribution of the ciphertext represents the letter `E` in the plaintext, then she can automatically deduce k. For instance, suppose that Eve determined the most frequent character in the ciphertext to be `O` $\leftrightarrow$ 14 as in Figure 2.4. She assumes this ciphertext character came from plaintext character `E` $\leftrightarrow$ 4. Then she deduces that $k = 14 - 4$, the difference between the final value and the assumed initial value.

There are limitations to this attack. The character `E` is only the most likely character in English text *on average*. In short pieces of plaintext, this property will likely not hold true, as you can convince yourself in any number of ways. We should only expect for this averaging argument to hold true for large pieces of ciphertext. This means that Eve may have to listen to the public channel for a while before being able to act on what she's learned.

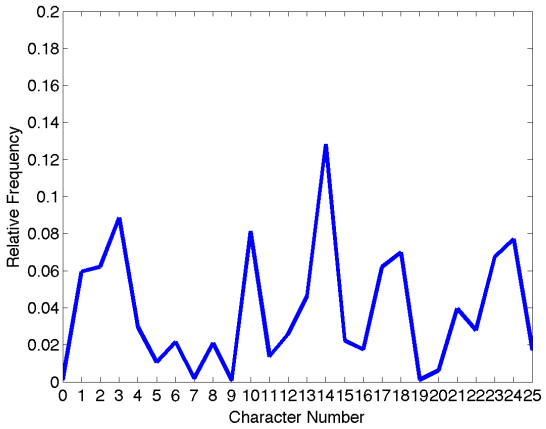

Figure 2.4: Relative frequency distribution for a long English novel after encryption with a Caesar cipher with key $k = 10$. Notice that the relative frequencies have been shifted to the right by 10 positions from Figure 2.3. In particular, E $\leftrightarrow$ 4 has become O $\leftrightarrow$ 14.

2.3 Affine Ciphers

Affine ciphers can be seen as a generalization of the Caesar ciphers presented in Section 2.2. Here, Alice and Bob first agree on a *pair* of numbers (k, ℓ) to use as the key. Using two keys instead of one substantially increases the size of the key space, and therefore increases the work Eve would need to do in order to break the cipher with a **brute force attack**. In this simple but often intensive attack, Eve would simply try to decipher the ciphertext using every key in the key space. The chances of producing legible plaintext when deciphering with the incorrect key is very small, so if Eve's efforts produce legible plaintext when deciphering with a particular key, she can be relatively certain that she has found the correct key.

2.3.1 Affine Encryption

Alice forms the ciphertext $\mathbf{c}$ from the plaintext $\mathbf{p}$ according to the following equation:

$$\mathbf{c} = (\ell\mathbf{p} + k) \bmod 26.$$

For a concrete example, imagine that Alice and Bob agree on the key $(3, 4)$. Then,

$$\texttt{A} \leftrightarrow 4(0) + 3 = 3,\ \texttt{B} \leftrightarrow 4(1) + 3 = 7, \ldots,\ \texttt{Z} \leftrightarrow 4(25) + 3 \equiv 25 \bmod 26.$$

Notice that unlike in a Caesar cipher, adjacent letters in the plaintext alphabet are not mapped to adjacent numbers in the ciphertext. In fact, if two letters are adjacent in the plaintext alphabet, then their numerical representations in the ciphertext differ by exactly ℓ.

2.3.2 Affine Decryption and Multiplicative Inverses

In order to recover the plaintext, Bob needs to undo whatever Alice has done in order to recover the plaintext $\mathbf{p}$ from the ciphertext $\mathbf{c}$. It may help to think of affine decryption symbolically. If Alice encrypts according to the function

$$\mathbf{c} = (\ell\mathbf{p} + k) \bmod 26, \tag{2.3.1}$$

then solving for $\mathbf{p}$ using traditional methods, Bob should compute

$$\mathbf{p} = \ell^{-1}(\mathbf{c} - k) \bmod 26. \tag{2.3.2}$$

We saw in the previous section that $\mathbb{Z}_{26}$ is a group under addition, and so the additive inverse $-k$ exists for any choice of key k. But what about ℓ^{-1}?

Traditionally, we have thought of the number x^{-1} as the number we would need to multiply by x in order to arrive at 1, the multiplicative identity, under "normal" arithmetic. For real numbers, we've always had $x^{-1} = 1/x$ for any nonzero x. We considered $x = 0$ to be the unique real number that had no multiplicative inverse.

The same general idea holds in modular arithmetic, but the specifics change. The inverse of an element x is still the element x^{-1} such that $x^{-1}x$ gives 1, but now all arithmetic is modular. We call an element x that has an inverse in $\mathbb{Z}_n$ a **unit** of $\mathbb{Z}_n$. For instance, in $\mathbb{Z}_{26}$, both the elements 3 and 9 are units, because

$$3 \cdot 9 = 27 \equiv 1 \bmod 26. \tag{2.3.3}$$

We conclude that $3^{-1} \equiv 9$ and $9^{-1} \equiv 3$. Similarly, we have

$$5 \cdot 21 = 105 \equiv 1 \bmod 26, \tag{2.3.4}$$

so that 5 and 21 are an inverse pair modulo 26. Notice that the element 1 is *always* a unit in modulo *any* n, since $1 \cdot 1 \equiv 1 \bmod n$ for any n.

But what about 2 modulo 26? We can begin by guessing and checking. Nothing unusual happens until we arrive at 13, for which we have

$$2 \cdot 13 = 26 \equiv 0 \bmod 26. \tag{2.3.5}$$

This is the type of thing that could *never* happen in real number arithmetic: we have the product of two nonzero numbers resulting in 0. We call any numbers that have this property **zero divisors**; for instance, 2 and 13 are zero divisors in $\mathbb{Z}_{26}$. What's even stranger, this property implies that 2 cannot have a multiplicative inverse modulo 26. We'll prove this claim in the next subsection; for now, let's continue thinking about the application at hand.

Imagine Alice and Bob agree on the key $(k = 4, \ell = 13)$. Notice that here $\ell = 13$ is a zero divisor in $\mathbb{Z}_{26}$. Notice what happens during encryption of `A` and `C`:

$$\texttt{A} \leftrightarrow 13 \cdot 0 + 4 = 4 \equiv 4 \bmod 26, \quad \texttt{C} \leftrightarrow 13 \cdot 2 + 4 = 30 \equiv 4 \bmod 26.$$

So if Bob receives a 4 in the ciphertext, it's impossible for him to tell whether Alice was encrypting `A` or `C`! This shows that we've been glossing over an important point when we've talked about Bob "undoing" whatever Alice did: Bob needs to be able to *uniquely* undo whatever Alice has done. If Alice uses a zero divisor for ℓ, Bob is left guessing what Alice meant. This is not a good situation!

We'll leave the decryption algorithm as an exercise, with the stipulation that Bob can assume that the key ℓ that Alice and Bob have agreed upon is not a zero divisor. We'll discuss a quick way in which Bob can actually compute the inverse ℓ^{-1} at the end of the next section.

2.3.3 Rings

With Caesar ciphers, we saw that $\mathbb{Z}_{26}$ was a group using addition modulo 26 as our operator for adding elements. But in affine ciphers, there's also a multiplicative operation that makes sense in this context. Here, we're dealing with a cousin of a group called a **ring**. Any ring R has the following properties:

- The elements of R form a commutative additive group.

- The elements of R are closed under multiplication, meaning that ab is an element of R for any choice of a and b.
- There is a multiplicative identity 1 such that $a \cdot 1 = 1 \cdot a = a$ for any a in R.
- Multiplication is associativity, *i.e.*, $a(bc) = (ab)c$ for any choice of a, b, c in R.
- Multiplication distributes over addition, meaning $a(b + c) = ab + ac$.

If in addition we have the property that $ab = ba$ for any choice of a and b in the ring, we say that the ring is **commutative**; notice that the elements in the equation move around or "commute." Most of the rings that we'll see over the course of our exploration will be commutative. We could verify that $\mathbb{Z}_n$ is a commutative ring for any choice of n; we will use this fact frequently. Most of the properties hold from our definitions of modular addition and multiplication.

The zero divisors and units provide an interesting structure to any ring. We'll use this opportunity to both illuminate this structure as well as introduce some basic proof techniques that will be useful throughout the course.

Theorem 2.3.1. *If an element x is a unit in $\mathbb{Z}_n$, then x is not a zero divisor in $\mathbb{Z}_n$.*

Proof. We will use a proof strategy here called **proof by contradiction**. We'll assume that an element x of $\mathbb{Z}_n$ is both a unit and a zero divisor, and show that this establishes conclusively that a deep contradiction would exist. We will conclude that an element cannot be both a zero divisor and a unit at the same time.

Suppose that an element x in $\mathbb{Z}_n$ is a unit. This implies by definition that there exists an inverse element x^{-1} such that $xx^{-1} \equiv x^{-1}x \equiv 1 \bmod n$. (All equivalences from here on in this proof will be modulo n without further mention.) Suppose now for contradiction that x is also a zero divisor. This implies, again by definition, that there exists some nonzero element y such that $yx \equiv 0$. Let's play around with these two facts and see what happens. Using the first assumption, we can write

$$xx^{-1} \equiv 1. \tag{2.3.6}$$

Then multiplying both sides by y gives

$$y(xx^{-1}) \equiv y \tag{2.3.7}$$

$$(yx)x^{-1} \equiv y \tag{2.3.8}$$

$$0 \equiv y, \tag{2.3.9}$$

where we've used the second assumption that x and y are zero divisors in the ring. Here we're seeing the contradiction first-hand: the element y cannot be zero, yet if x is to be both a zero divisor and a unit, then y must be equivalent to 0. We conclude that x could not have been a zero divisor and a unit in the first place. Therefore, if x is a unit, it is not a zero divisor, and vice versa. □

So we know if x is a unit, then x is not a zero divisor. But what if x is not a unit? We can cover this case, too.

Theorem 2.3.2. *If an element x is not a unit in $\mathbb{Z}_n$, then x is a zero divisor in $\mathbb{Z}_n$.*

Proof. Here we will use a proof strategy called the **pigeonhole principle**. The name might be silly, but the idea is rather foundational: if we have n items (or pigeons) to be put into $m < n$ locations (or pigeonholes), then at least 1 of the locations must contain at least 2 items (at least 1 pigeonhole must contain at least 2 pigeons). This tactic comes up over and over again in mathematics.

Suppose that a nonzero element x is not a unit of $\mathbb{Z}_n$. Then there does *not* exist a unit x^{-1} such that $xx^{-1} \equiv 1 \bmod n$. (Again, we will assume from here on out in the proof that all equivalences are taken modulo n.) Consider the product of x with *every* other element in $\mathbb{Z}_n$. There are n total products having the form

$$xr_1,\ xr_2,\ xr_3,\ \ldots, xr_n, \tag{2.3.10}$$

where the r_i are serving as our placeholders for the elements of the ring $\mathbb{Z}_n$. Since x is not a unit, none of these products is equivalent to 1. So there are at most $n-1$ values in $\mathbb{Z}_n$ that these products can assume. But there are n total products. By the pigeonhole principle, at least two of the products must be equivalent, allowing us to write

$$xr_i \equiv xr_j \tag{2.3.11}$$

for some $r_i \neq r_j$. We can rearrange this equivalence using the properties of rings.

$$xr_i - xr_j \equiv 0 \tag{2.3.12}$$

$$x(r_i - r_j) \equiv 0. \tag{2.3.13}$$

Since $r_i \neq r_j$, we have $(r_i - r_j) = y \neq 0$. But this implies that $xy \equiv 0$ with neither x nor y equal to 0. We conclude that if x is not a unit, then it must be a zero divisor. □

We can combine the previous two results into a simple but powerful statement about the structure of $\mathbb{Z}_n$ (or indeed, any ring).

Theorem 2.3.3. *An element x of $\mathbb{Z}_n$ is either a unit or a zero divisor but not both.*

The fact that the elements of a ring can be partitioned into two disjoint sets, one of zero divisors and one of units, is nice, but rather theoretical. For instance, for a given x, how can we tell if x is a zero divisor or a unit? Fortunately, there exists a simple test related to the concept **greatest common divisor**. The greatest common divisor (also known as **greatest common factor**) of two numbers x and y (often abbreviated $\gcd(x, y)$) is the largest number d that

divides both x and y evenly. We can make some direct observations including $\gcd(x, y) \leq x$ and $\gcd(x, y) \leq y$. We also have $\gcd(x, y) \geq 1$, because every number is divisible by 1. In Matlab, $\gcd(x, y)$ is implemented as `gcd(x,y)`.

Example 2.3.1. *The greatest common divisor of 4 and 42 is 2. We can list of the divisors of 4 (1,2, and 4) and the divisors of 42 (1, 2, 6, 7, 21, 42). The greatest number appearing in both lists is 2. Therefore* $\gcd(4, 42) = 2$. *We can easily confirm this fact in Matlab:*

```
>> gcd(2,42)

ans =

     2
```

We say that two integers x and n are **coprime** if they share no common factor other than 1. Said another way, we call x and n coprime if $\gcd(x, n) = 1$. This notion of coprimality is intimately connected to the existence of a multiplicative inverse of x in $\mathbb{Z}_n$.

Theorem 2.3.4. *If $gcd(x, n) = 1$, then x is a unit of $\mathbb{Z}_n$.*

Proof. We will use a proof strategy that we'll refer to as **standing on the shoulders of giants**, which means that we will use a result from another mathematician to help us prove the desired result. This strategy is crucial for the development of mathematics over time. The community feeds itself in a way, with the work of earlier researchers paving the way for new work.

Let's suppose that $\gcd(x, n) = 1$. We'll show that x must have an inverse in $\mathbb{Z}_n$. The quickest way is to use a result called **Bézout's identity** which says that for any x and n, we can find integers a and b such that

$$ax + bn = \gcd(a, n). \tag{2.3.14}$$

(Notice that either a or b must be negative.) Before we go on, let's see an example of Bézout's identity. In the previous example, we showed that $\gcd(4, 42) = 2$. We could confirm that

$$11(4) - 1(42) = 2. \tag{2.3.15}$$

Notice that Bézout's identity doesn't tell us exactly what values a and b will take, it just guarantees that such a pair of numbers must exist.

In terms of our work here, we have

$$ax + bn = 1 \tag{2.3.16}$$

But since any multiple of n is congruent to 0 modulo n, we have $bn \equiv 0 \bmod n$, and so

$$ax \equiv 1 \bmod n. \tag{2.3.17}$$

This directly implies that a is the multiplicative inverse of x in $\mathbb{Z}_n$. We conclude that if $gcd(x, n) = 1$, then x is a unit in $\mathbb{Z}_n$. □

So if $gcd(x, n) = 1$, then x has an inverse modulo n. But what about if $gcd(x, n) > 1$?

Theorem 2.3.5. *If $gcd(x, n) = d > 1$, then x has no inverse in $\mathbb{Z}_n$.*

Proof. Here we will use a strategy we might call **bootstrapping** which involves using an earlier result that we ourselves have shown in order to get to the desired result. Bootstrapping is an integral part of tackling big problems, because interesting results are often too difficult to prove in a single sweep; we need to prove several small things and then assemble them into the larger picture.

Suppose we have an element x in the ring $\mathbb{Z}_n$. By definition, $x < n$. Furthermore, let's assume that $\gcd(x, n) = d > 1$. This implies that $x = dq_x$ for some q_x, and $n = dq_n$ for some q_n. We can combine these facts with some of our commutative ring properties to show that x must be a zero divisor:

$$q_n x \equiv q_n(dq_x) \equiv (dq_n)q_x \equiv nq_x \equiv 0 \bmod n. \tag{2.3.18}$$

Since x is a zero divisor, our earlier result from Corollary 2.3.3 states that x is not a unit. Therefore, we can conclude that x has no inverse in $\mathbb{Z}_n$. □

We can combine our results into another simple and powerful statement about inverses (or lack thereof) in $\mathbb{Z}_n$.

Theorem 2.3.6. *An element x has an inverse in $\mathbb{Z}_n$ if and only if $\gcd(x, n) = 1$.*

In terms of affine ciphers, our results imply that if $\gcd(\ell, 26) = 1$, then ℓ is an appropriate key for Alice and Bob to use, and if $\gcd(\ell, 26) > 1$, then ℓ is not appropriate. We can use the supplied function `modInv(x,n)` to find the multiplicative inverse of x in $\mathbb{Z}_n$ in Matlab:

```
>> modInv(5,11)

ans =

     9

>> modInv(5,26)

ans =

    21
```

Notice that the number 5 has different inverses in different rings: $5^{-1} \equiv 9$ in $\mathbb{Z}_{11}$ and $5^{-1} \equiv 21$ in $\mathbb{Z}_{21}$.

2.3.4 Affine Cryptanalysis

While larger than the key space of Caesar ciphers, the key space of affine ciphers is still well within the bounds of a simple **brute force attack**. Here, Eve can

simply cycle through the pairs (k, ℓ) and attempt decryption. If the result is a clear plaintext, then Eve has overwhelmingly likely stumbled upon the correct key. This is not an especially elegant or insightful attack, but in the absence of any better idea, this is Eve's default attack option.

In terms of a Matlab implementation, Eve could search the key space with a *double* `for` loop. In pseudo code, her function might look something like

```
function affineDecrypterBruteForce(ciphertext)
    for k from 1 to 25
         for ell from 1 to 25
             if gcd(ell,26) == 1
                 affineDecrypt(ciphertext, k, ell)
             end
         end
     end
end
```

There are a couple of things to note here. First notice that the inner `for` loop runs in its entirety for each value of `k` in the outer loop. So the (k, ℓ) pairs tested would proceed in order as

$$(1,1),\ (1,2),\ (1,3),\ \ldots,\ (1,25),\ (2,1),\ (2,2),\ \ldots,\ (2,25),\ \ldots,(25,24),\ (25,25).$$

Second, notice that Eve only bothers checking values of `ell` that she knows have a multiplicative inverse in $\mathbb{Z}_{26}$.

2.4 Polyalphabetic Ciphers

A major weakness of many classical cryptosystems is the limited key space. Even without advanced computational technology, Eve would be more than capable of performing a straight-forward brute force attack on an affine cipher. Despite their weaknesses, these secrecy technologies were the state of the art until the early Renaissance period when a collection of techniques called polyalphabetic ciphers were developed. Here, Alice and Bob agree on a *string* of characters to use as the key. As we'll see, enlarging the key from a single character to a word or phrase can form dramatically larger key spaces, providing a much high degree of security from simple brute force attacks. This process will culminate in a discussion of one-time pads, a cryptosystem that, when properly implemented, Eve could not break even with enormous time and effort.

2.4.1 Encryption

Alice and Bob begin by agreeing on a key, for instance the word "KEY." Alice lines up her plaintext, for instance "PLAINTEXT", together with a string which repeats the key:

```
PLAINTEXT
KEYKEYKEY
```

Alice then forms the ciphertext by vertically adding the numerical representations of the two strings modulo 26.

$$\begin{array}{rccccccccc} & 15 & 11 & 0 & 8 & 13 & 19 & 4 & 23 & 19 & \\ + & 10 & 4 & 24 & 10 & 4 & 24 & 10 & 4 & 24 & \\ \hline & 25 & 15 & 24 & 18 & 17 & 43 & 14 & 27 & 43 & \\ \equiv & 25 & 15 & 24 & 18 & 17 & 17 & 14 & 1 & 17 & \mod 26 \end{array} \tag{2.4.1}$$

We call this type of addition **component-wise**, since the sum is obtained by adding like components in both vectors. So Alice sends the ciphertext `ZPYSRROBR` corresponding to the component-wise sum of the plaintext and the repeated key. Notice that the length of the ciphertext does not have to be a multiple of the length of the key. For example, Alice could encrypt her name by adding vertically in the following array:

```
ALICE
KEYKE
```

Notice that adjacent characters in the plaintext are encrypted in different ways. We call this type of technique **polyalphabetic**, because the same letter in the plaintext can be mapped to different letters in the ciphertext. In effect, each letter in the key induces a new Caesar cipher alphabet to be used. The particular type of polyalphabetic cryptosystem Alice has used here is called the **Vigenère cipher**. Later, we'll see even more secure variants.

Implementing this encryption routine in Matlab is most easily accomplished through the use of a specialized function called `repmat`:

```
>> help repmat
 REPMAT Replicate and tile an array.
    B = repmat(A,M,N) creates a large matrix B consisting of an M-by-N
    tiling of copies of A. The size of B is [size(A,1)*M, size(A,2)*N].
    The statement repmat(A,N) creates an N-by-N tiling.

>> repmat([10 4 24],1, 3)

ans =

    10     4    24    10     4    24    10     4    24
```

Suppose we already have a variable `p` which is a numeric vector representing the plaintext and another variable `k` which is a numeric vector representing the key. Our encryption function might look something like:

```
function c = vigenereEncrypter(p,k)
    numReps  = ceil(numel(p)/numel(k));
    repeatedKey  = repmat(k,1,numReps);
    c   = mod(p + repeatedKey(1:numel(p)),26);
end
```

The first line of the function determines the number of times we need to repeat the key. For instance, in the example above, the plaintext `ALICE` has 5 characters, so `numel(p)` would be 5, and the key `KEY` has 3 characters, so `numel(k)` would be 3. Then `ceil(numel(p)/numel(k))` would equal 2; recall that ceiling is the equivalent of rounding up.

The second line in the function forms the repeated key. At this point in the function, the value of repeated key would be the numeric representation of the string `KEYKEY`.

The third line of the function forms the ciphertext. Notice that in the command `repeatedKey(1:numel(p))` we **slice** the vector `repeatedKey` to use only the portion that corresponds to positions in the plaintext. (More generally, slicing refers to any action where we isolate part of a vector or matrix, *e.g.* `x(3:10)`.)In terms of the running example, we only add the numeric representation of `KEYKE`, since `numel(p) = 5`.

2.4.2 Decryption

Bob's decryption process involves subtracting (modulo 26) from each component of the ciphertext the value that Alice added. This again would involve an execution of `repmat`. We'll leave this as an exercise.

2.4.3 Cryptanalysis

The cryptanalysis of poylalphabetic ciphers is much more difficult than that of other classical ciphers exactly because the key space is so much larger. A simple

brute force attack cannot be expected to be fruitful against a Vigenère cipher with moderately long key length. For instance, even if Even knows that the key has length 10, she would have $26^{10} \approx 1.4 \times 10^{14}$ possible keys to test. As the key length grows, this task becomes infeasible. Notice, too, that increasing the key length doesn't harm Alice or Bob at all. Eve will need to be more clever in her attack.

Eve has two main tasks when confronting a piece of Vigenère ciphertext:

(1) determine the length of the key

(2) determine what the key is

With the key in hand, Eve can completely decrypt the ciphertext.

Key Length

In order to determine the key length, Eve can employ a tactic called **Friedman analysis**. In the early 1900s, William Friedman began thinking about *coincidences*, defined here as the chance event that the the same position in two strings of text contain the same letter. Between two uniformly random strings of English text, this chance is rather low. (You could convince yourself it is in fact exactly 1/26.) But Friedman's real contribution was considering coincidence in the case where the two strings are not uniformly random, but in fact related to one another.

Imagine Eve compares the intercepted ciphertext with a shifted version of itself and looks for coincidences, as seen below:

```
SXUKWQOZCXSAVSAUSDKZCBCUKVKOZCXMLQMLDLCcICYRCOLGVP...
_SXUKWQOZCXSAVSAUSDKZCBCUKVKOZCXMLQMLDLcCICYRCOLGV...
```

The underscore character here simply denotes the shift, and lowercase letters denote coincidences. With one shift, Eve observes only a single coincidence in the first 50 characters or so. This might not be too surprising, because in order for two adjacent letters in the ciphertext to be the same (which would lead to a coincidence after a single shift), the key and the plaintext would have to interact in a very peculiar way. We'll delve deeper into this later. Eve can trying shifting again, for a total of 2 shifts, and looking for coincidences:

```
SXUKWQOZCXSAVSAUSDKZCBcUKVkOZCXMLQMLDlCCIcYRCOLGVP...
__SXUKWQOZCXSAVSAUSDKZcBCUkVKOZCXMLQMlDLCcICYRCOLG...
```

Here Eve sees 4 coincides. She can repeat the process for a total of 3 shifts:

```
SXUKWQOZCXSAVsaUsDKZCBCUKVKOZCXMLQmlDLCClcYRcOLGVP...
___SXUKWQOZCXsaVsAUSDKZCBCUKVKOZCXmlQMLDlcCIcYRCOL...
```

Here, Eve sees 8 coincidences! This is *twice* as many as any shift so far. In a larger chunk of ciphertext, the difference between the coincidences would be even higher. Finally, if she shifts a total of 4 positions, she again observes a small number of coincidences; here, in fact, she observes none:

```
SXUKWQOZCXSAVSAUSDKZCBCUKVKOZCXMLQMLDLCCLCYRCOLGVP...
____SXUKWQOZCXSAVSAUSDKZCBCUKVKOZCXMLQMLDLCCLCYRCO...
```

But what's actually going on here? When Eve shifts a multiple of the key length, the two characters vertically aligned with one another were encrypted using the same character of the key. So if the two plaintext characters started equal, then their ciphertext characters were equal, too. This sort of coincidence is much more likely than two characters that were encrypted using different Caesar ciphers agreeing by chance. (We could prove this much more formally as Friedman did, but it would be relatively involved; let's just stick with the intuition.) Therefore, when Eve shifts by a multiple of the key length, she should expect to see many more coincidences than when she doesn't.

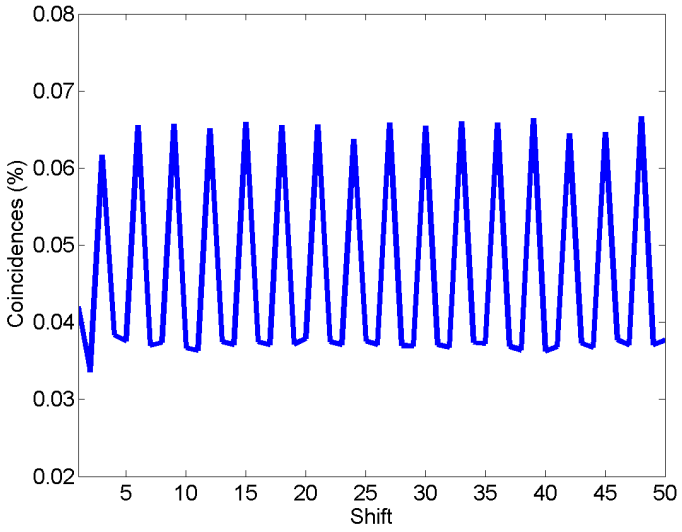

Figure 2.5: Friedman analysis for a large piece of Vigenère ciphertext. Spikes occur at multiples of the key length. Here it seems that the key is likely of length 3.

Imagine Eve automates this process, shifting and keeping track of the percentage of positions that were coincidences. In Matlab, for vectors `x` and `y`, we can define a new vector `z = (x == y)` that is 1 in positions in which `x` and `y` agree, and 0 otherwise. Notice that since single `=` indicates assignment, we need to use `==` to indicate the equality in the traditional sense of the word. Eve could then count coincidences in two vectors by summing: `sum(x == y)`.

The results of such an endeavor can be seen in Figure 2.5. Notice that the "background level" of coincidences if roughly $1/26 = 3.85\%$, the proportion we would expect if the two strings were uniformly random. Out of the background noise, we see huge spikes at regular intervals. These are the coincidences caused

by a shift that is equal to a multiple of the key length. Since the spikes occur every 3 shifts in the Figure 2.5, we conclude that the length of the key is most likely in this piece of ciphertext.

Key Contents

Even with the the length of the key in hand, Eve still has a tough job ahead. After all, if the key has length 100, there would be 26^{100} possible keys to search if she were performing a brute force attack. Fortunately for Eve, she can be much more clever if the ciphertext is large enough. For instance, she knows that each character in the plaintext is encrypted using a Caesar cipher, with the particular number of shifts determined by the value of the key in a particular position. Once Eve knows the length of the key, she can group all ciphertext characters that were encrypted with the same Caesar cipher and form a frequency distribution of the characters. For instance, after Eve knows the ciphertext `c` was encrypted using a key of length 3, she knows the elements `c(1:3:end)` were all encrypted using the same Caesar cipher. For instance the command `frequency(c(2:3:end))` would show Eve the relative frequency distribution of all ciphertext characters encrypted using the *second* character in the key.

Eve can perform a similar process for each of the positions in the key. Note that this holds better for long ciphertexts and worse for short ciphertexts. For short ciphertexts, we should expect that our best guess will often be wrong, because the frequencies in a short plaintext do not necessarily agree very well with the frequencies in English plaintext at large. Even still, Eve has drastically narrowed the field of possible choices.

2.4.4 Polyalphabetic Variants

If Alice and Bob use a polyalphabetic cipher with key length $\ell = 1$, they have essentially agreed to use a Caesar cipher of some variety. There are only 26 possible keys, and Eve could easily test each in order to see if the candidate plaintext makes sense. Alice and Bob can begin to lengthen the key in any number of ways.

A **running key cipher** is a polyalphabetic cipher in which the key is the same length as the plaintext and is drawn from an agreed upon book, magazine, or the like. These cryptosystems were relatively popular in the Cold War. Covert field agents would be supplied with a common and innocuous book, a farmer's almanac or popular book of fiction, for example. To encrypt a message, agents would simply begin using the book itself as the key. Since the plaintext would be drastically shorter than a key, there would be no need to repeat the key. With no repetition, the cryptanalytic strategy outlined previously no longer applies. This provided relatively strong security at relatively low cost, a good combination for most applications. That being said, using English text as a key is not perfect. Certain character combinations (*e.g.*, `TE`, `RE`, `TH`, *etc.*) are much more common than others (*e.g.*, `QU`, `CZ`, `PH`). Similar facts hold for character strings of length 3, 4, and so on. An experienced cryptanalyst can use these

properties to intelligently narrow her search. Moreover, English is often highly structured. If a cryptanalyst begins to figure out chunks of the key, it can be easy to begin guessing the remainder from context. We might call this the "Wheel of Fortune effect."

One-time pads are the most secure classical cryptosystems. Here, like with a running key cipher, the key is the same length as the plaintext. The major difference is that in a one-time pad, each character is generated uniformly randomly and independently from all other key characters. Once a key is used, it is never reused; this is the origin of the "one-time" moniker. Properly implemented, one-time pads provide the highest level of security possible: Eve gets no new information about the plaintext from the ciphertext. But nothing is ever free. While running pads are easy to use, one-time pads are incredibly difficult. Alice and Bob need a common, large source of uniformly random characters or numbers. While large sources of text are easy to find, large random samples are virtually relatively rare. Moreover, since Alice and Bob can never reuse key while maintaining the highest level of secrecy, they need a way to replenish their key stock, presumably while physically separated from one another. These implementation issues make the perfect cryptosystem in theory used very rarely in practice.

Chapter 3

Modern Cryptosystems

Modern cryptosystems are designed to disguise plaintext composed of 0s and 1s. These binary digits, also known as bits, are the most basic unit of information and the foundational unit of memory in modern computer hardware. Most modern cryptosystems rely on the assumed difficulty of certain mathematical problems related to arithmetic in $\mathbb{Z}_n$. After detailing the most commonly used modern cryptosystems, we will widen our scope by investigating the concept of a cryptographic hash function. Combining modern encryption and hash functions will allow us deal with state-of-the-art applications such as authentication and zero-knowledge proofs.

3.1 Diffie-Hellman Key Exchange

One of the major problems with classical cryptosystems is the idea of a key shared between Alice and Bob. Each cryptosystem seen in Chapter 2 relied on some piece of information being jointly accessible to Alice and Bob but not to Eve. While this is easy to assume in theory, it's much more difficult to implement in practice. After all, if Eve is monitoring the public channel, how do Alice and Bob securely agree upon a key? In many cases, the only reasonable option was a physical exchange. For instance, before a field agent was sent abroad, she might be given a copy of a particular book to use in a running key cryptosystem. In a more extreme example, physical copies of cryptographic keys could be placed in a briefcase which is then handcuffed to a courier. Whatever the methodology, exchanging keys in the classical cryptographic era was expensive, time-consuming, and dangerous. Key exchange in the modern era is much easier and safer, but much more mathematically interesting, than previous methods.

3.1.1 Primitive Roots

Rings in general have many interesting properties that can be exploited in cryptography. One such notion is primitive roots. We say an element z of $\mathbb{Z}_n$ is a **primitive root** (or simply is primitive) if every element x of $\mathbb{Z}_n$ that is coprime to n can be written as $x \equiv z^k$ for some integer k. Said another way, an element z is primitive in a ring if every unit is equivalent to a power of z.

Example 3.1.1. *The element 2 is primitive in $\mathbb{Z}_5$. By taking consecutive powers of 2, we can see that all elements in the group are accounted for*

$$2^1 \equiv 2,\ 2^2 \equiv 4,\ 2^3 \equiv 3,\ 2^4 \equiv 1.$$

Example 3.1.2. *The element 2 is not primitive in $\mathbb{Z}_6$. By taking consecutive powers of 2, we can see that some elements that are coprime to 6 are left out, for instance 1, 3, and 5.*

$$2^1 \equiv 2,\ 2^2 \equiv 4,\ 2^3 \equiv 2, \ldots$$

Example 3.1.3. *The element 3 is not primitive in $\mathbb{Z}_{11}$. By taking consecutive powers, we can see that some elements that are coprime to 11 are left out, for example 2:*

$$3^1 \equiv 3,\ 3^2 \equiv 9,\ 3^3 \equiv 5,\ 3^4 \equiv 4,\ 3^5 \equiv 1, \ldots$$

Example 3.1.4. *The element 6 is primitive in $\mathbb{Z}_{11}$. By taking consecutive powers, we can see that some elements that are coprime to 11 are left out, for example 2:*

$$6^1 \equiv 6,\ 6^2 \equiv 3,\ 6^3 \equiv 7,\ 6^4 \equiv 9,\ 6^5 \equiv 10,\ 6^6 \equiv 5,\ 6^7 \equiv 8,\ 6^8 \equiv 4,\ 6^9 \equiv 2,\ 6^{10} \equiv 1$$

As we can see in these examples, as we increase the value of the exponent k, the value of x^k seems to jump all over the range. Said another way, given an

element y and a primitive element x, it's very difficult guess what value of k leads to $y \equiv x^k$. This is known as the **discrete logarithm problem**. The security of several different cryptographic protocols rests on the difficulty of this problem. There is no *proof* that this problem is hard, however; it could be the case that tomorrow someone invents a clever new way to crunch discrete logarithms, and these cryptosystems would fall apart. That being said, these techniques have been around for some time, and thus far no one has made public a cryptanalytic strategy that would solve discrete logarithms in a reasonable amount of time.

3.1.2 Key Exchange

Alice and Bob will each keep track of several pieces of information over the course of the Diffie-Hellman key exchange. First, Alice and Bob agree on a large prime number p and a primitive element g of $\mathbb{Z}_p$. In practice, p could have *hundreds* of digits in its decimal representation. Alice and Bob make these numbers public. In additional, Alice keeps a private key a (which she does not share with anyone), and Bob keeps a private key b (which, similarly, he does not share with anyone). Notice that already this formulation is different from classical cryptosystems, in that Alice and Bob publicize some information openly.

Alice and Bob will both compute a shared number $s \equiv g^{ab}$, but in a way that Eve will not be able to replicate. Alice and Bob could then use s as the key for secure classical cryptographic communications.

Alice begins the exchange by computing $A \equiv g^a \bmod p$ and sending A over the public channel to Bob. Notice that because of the difficulty of the discrete logarithm problem, Eve cannot determine Alice's private key a, even with knowledge of A and g! When Bob receives A, he can compute the shared secret by exponentiating:

$$s \equiv A^b \equiv (g^a)^b \equiv g^{ab} \bmod p.$$

Similarly, Bob computes $B \equiv g^b \bmod p$ and sends B over the public channel to Alice. Again, Eve cannot figure out b given B and g due to the difficulty of the discrete logarithm problem. When Alice receives B, she can compute the shared secret in much the same way Bob did:

$$s \equiv B^a \equiv (g^b)^a \equiv g^{ab} \bmod p.$$

So Alice and Bob have computed a shared secret number s in $\mathbb{Z}_p$ while only transmitting harmless information across the public channel.

Example 3.1.5. *Alice and Bob agree on a prime number $p = 11$ and a primitive element $g = 2$. Alice randomly generates a random key $a = 5$, and Bob randomly generates his private key $b = 9$. Alice and Bob exchange*

```
>> A = mod(2^5,11)

A =
```

```
     10


>> B = mod(2^9,11)

B =

     6
```

Alice and Bob can the compute the shared secret $s = 10$ *via* $s \equiv B^a$ *and* $s \equiv A^b$*, respectively:*

```
>> s = mod(6^5,11)

s =

    10

>> s = mod(10^9,11)

s =

    10
```

3.1.3 Matlab Implementation

Many of the difficulties in implementing cryptography can be attributed to translating pure mathematical ideas to some sort of computational representation. One such example can be found in Matlab's `mod` command. Take for instance,

```
>> mod(2^100,3)

ans =

     0
```

After a moment of thought, we can conclude that there is no way this answer is right! After all, 2^{100} can't be divisible by 3, exactly because it's a power of 2. So what's going on here? The number 2^{100} is so big that Matlab cannot accurately represent it as an integer; it lops off some of the less significant digits in order to hold the quantity in memory. This is fine most of the time, but not in our case. For our cryptographic techniques to work, we need full precision.

There are techniques for dealing with arithmetic operations on large numbers. The function `powMod` included with these notes, for instance, is able to compute $g^x \bmod n$ for large g, x and n:

```
>> help powMod
  powMod(BASE,EXPONENT,MODULO) computes BASE^EXPONENT mod MODULO using the
  right-to-left binary method.
```

The exact method used to perform this computation isn't really that important, though it is very interesting if you're so inclined. The good news is that we can accurately compute modular quantities with large arguments:

```
>> powMod(2,100,3)

ans =

     1
```

We could confirm that this answer is correct by considering $2^{100} = 4^{50}$. Then since $(ab) \bmod n \equiv (a \bmod n)(b \bmod n)$, we have

$$4^{50} \equiv (4 \bmod 3) \cdots (4 \bmod 3) \equiv 1.$$

Of course, this is just one example, but it is heartening to see everything turn out the way we should expect.

We can see an example of `powMod` used in the Diffie-Hellman scheme.

Example 3.1.6. *Alice and Bob agree on a prime number $p = 509$ and a primitive element $g = 2$. Alice randomly generates a random key $a = 72$, and Bob randomly generates his private key $b = 215$. Alice and Bob exchange, respectively,*

```
>> A = powMod(2,72,509)

A =

   453
>> B = powMod(2,215,509)

B =

   249
```

They can then both compute the shared secret:

```
>> s = powMod(249,72,509)

s =

   281
>> s = powMod(453,215,509)

s =

   281
```

3.1.4 Man-in-the-middle Attacks

While the difficulty of the discrete logarithm makes it tough for Eve to outright break the Diffie-Hellman key exchange scheme, a simple but effective technique known as a **man-in-the-middle** attack can be quite effective. Imagine that Alice and Bob are trying to perform a secure key exchange. A malicious adversary, traditionally named Mallory, puts herself between the two parties by pretending to be Alice when communicating with Bob, and pretending to be Bob when communicating with Alice. To Alice and Bob, everything looks normal. Little do they know, Mallory is privy to their entire conversation.

For an example, imagine that Alice and Bob want to exchange a key. They publish their chosen prime p and primitive element g, as well as their public keys $A \equiv g^a \bmod p$ and $B \equiv g^b \bmod p$. Mallory has rightful access to all this information, but does not have access to the private keys a and b. Mallory choses her own private m. She sends $g^m \bmod p$ to Alice pretending to be Bob and sends $g^m \bmod p$ to Bob pretending to be Alice. In this way, she sets up a shared secret $s_a \equiv g^{ma} \bmod p$ with Alice and a shared secret $s_b \equiv g^{mb} \bmod p$ with Bob. Imagine now that Alice wants to send a secure message to Bob. She encrypts the message using the shared secret s_a (which she thinks she's agreed upon with Bob). Mallory intercepts the message and decrypts it using s_a. To preserve the ruse, she then *re-encrypts* the plaintext using the key s_b and passes the resulting ciphertext along to Bob.

To Alice and Bob, it appears as if nothing out of the ordinary has happened here. What's worse, Mallory can continue this strategy indefinitely. In fact, she must. If Mallory were to miss a message, Bob would receive a message from Alice that had been encrypted with key s_a, not with key s_b as he'd expected.

One way that Alice and Bob could realize that Mallory was intercepting their traffic would be to examine how long it takes between Alice's transmission and Bob's reception. This is period is known as **latency**. If the latency is much longer than Alice or Bob would've expected given conditions on the public channel, then there may be a man-in-the-middle attack in progress. A simple but relatively expensive technique builds off this concept. Alice and Bob agree that in each transmission they will send the result of an time-consuming computation that the other can verify; we'll see a collection of specific examples if Section 3.3. If Alice and Bob notice that the latency is much higher than network conditions would predict, they can be relatively sure that Mallory is delaying the message's arrival as she spends time on the required computation.

3.2 RSA

The RSA cryptosystem is perhaps the most widely used mathematical security technology in the world. Rivest, Shamir, and Adelman invented the technique in the late 1970s, only shortly after Diffie and Hellman introduced the world to the idea of public key cryptography. While Diffie-Hellman only allows Alice and Bob to agree on a shared secret, RSA allows Alice and Bob to actually exchange messages. This increase in utility comes at the cost of increased mathematical complexity. There are several ideas that we need to master before we can discuss Rivest, Shamir, and Adelman's breakthrough in earnest. The first is a rather old piece of mathematical technology: Euler's totient function.

Definition 3.2.1. *Euler's totient function $\phi(n)$ is defined as the number of positive integers less than n which are coprime to n.*

Example 3.2.2. *We have $\phi(6) = 2$ since only 1 and 5 are coprime to 6.*

Example 3.2.3. *We have $\phi(9) = 6$ since only 1, 2, 4, 5, 7, and 8 are coprime to 9*

Example 3.2.4. *We have $\phi(11) = 10$ since all integers between 1 and 10 are coprime to 11.*

In the codebase provided with these notes, there is an implementation of Euler's totient function. Hopefully, it works in much the way you'd expect:

```
>> totient(9)
ans =
     6
```

Since the elements that are coprime with n are exactly the units of $\mathbb{Z}_n$, we have the following result, which is essentially just a conversion of terms.

Theorem 3.2.1. *There are $\phi(n)$ units in $\mathbb{Z}_n$.*

This idea may seem simple if you've had a bit of experience with multiplicative inverses in $\mathbb{Z}_n$, but it's actually critical to RSA. Since every element of $\mathbb{Z}_n$ is either a unit or a zero divisor, we can use the preceding idea to conclude that there are $n - \phi(n)$ zero divisors in $\mathbb{Z}_n$. So to compute $\phi(n)$, we can count either units or zero divisors, whichever is easier. We can use this strategy to prove one of the building blocks in RSA's security.

The totients of primes and products of primes are at the heart of RSA. For the case of a single prime, the totient isn't much more than a conversion between definitions.

Lemma 3.2.1. *For any prime p, we have $\phi(p) = p-1$, since all integers between 1 and $p-1$ are coprime with p by definition.*

Lemma 3.2.2. *If p and q are distinct primes, then*

$$\phi(pq) = \phi(p)\phi(q) = (p-1)(q-1).$$

Before we dive into the general proof, let's see a specific example.

Example 3.2.5. *We have $\phi(15) = \phi(3 \cdot 5) = (3-1)(5-1) = 8$. To see this, let's find all the zero divisors of $\mathbb{Z}_{15}$. An element of $\mathbb{Z}_{15}$ will be a zero divisor if it is either a multiple of 3 or a multiple of 5. There are 5-1 = 4 multiples of 3 that are less than 15, namely 3, 6, 9, and 12, and there are 3-1 = 2 multiples of 5 less than 15, namely 5 and 10. So there are $(3-1)+(5-1)$ zero divisors in $\mathbb{Z}_{3\cdot 5}$, which implies that there are $3 \cdot 5 - (3-1) - (5-1) - 1 = 15 - 2 - 4 - 1 = 8$ units. The final -1 accounts for the zero element.*

The general proof follows in exactly the same fashion from the previous example. We only really need to exchange $3 \leftrightarrow p$ and $5 \leftrightarrow q$.

Proof. A nonzero element in $\mathbb{Z}_{pq}$ will be a zero divisor if and only if it is either a multiple of p or a multiple of q. There are $p-1$ nonzero multiples of q which are less than pq, namely

$$q,\ 2q, \ldots,\ (p-1)q$$

and $q-1$ nonzero multiples of p which are less than pq, namely

$$p,\ 2p, \ldots,\ (q-1)p.$$

Since every nonzero element that is not a zero divisor is a unit, there are a total of

$$pq - (p-1) - (q-1) - 1 = pq - p - q + 1 = (p-1)(q-1)$$

units in $\mathbb{Z}_n$. The final -1 on the left side accounts for the zero element. We conclude that $\phi(pq) = (p-1)(q-1)$. □

Euler's totient function is deeply connected to the idea of order. Like the totient function, it may be a concept that you have seen bits and pieces of over the course of other investigations. It's worth taking the time to really dig into it in order to make the ideas of RSA more clear.

Definition 3.2.2. *The order of an element x in $\mathbb{Z}_n$ is the smallest nonzero value of k such that $x^k \equiv 1 \bmod n$. If no such value of k exists, then we say x has order ∞ in $\mathbb{Z}_n$.*

Example 3.2.6. *The element 3 has order 6 in $\mathbb{Z}_7$, because*

$$3^1 \equiv 3,\ 3^2 \equiv 2,\ 3^3 \equiv 6,\ 3^4 \equiv 4,\ 3^5 \equiv 5,\ 3^6 \equiv 1.$$

Example 3.2.7. *The element 2 has order 3 in $\mathbb{Z}_7$, because*

$$2^1 \equiv 2,\ 2^2 \equiv 4,\ 2^3 \equiv 1.$$

Example 3.2.8. *The element 2 has order ∞ in $\mathbb{Z}_6$, because*

$$2^1 \equiv 2,\ 2^2 \equiv 4,\ 2^3 \equiv 2, \ldots.$$

In the codebase provided with these notes, there is a function `ord(x,n)` which computes the order of an element x in $\mathbb{Z}_n$. We can confirm that this function gives us the values we found by hand earlier.

```
>> ord(2,7)
ans =
     3
>> ord(2,6)
ans =
   Inf
```

One of the most surprising facts about modern cryptography is that many of the results on which the field rests are quite old. Lagrange's theorem and Euler's theorem weave together the concepts of order of an element x in $\mathbb{Z}_n$ and the totient of n.

Result 3.2.1. *(Lagrange's theorem, proved in 1771) The multiplicative order of every unit x in $\mathbb{Z}_n$ divides $\phi(n)$.*

In Example 3.2.6, the order of 3 in $\mathbb{Z}_7$ is 6 which divides $\phi(7) = 6$. Similarly, in Example 3.2.7, the order of 2 in $\mathbb{Z}_7$ is 3 which divides $\phi(7) = 6$.

Theorem 3.2.2. *(Euler's theorem, proved in 1763) If a is coprime to n, then $a^{\phi(n)} \equiv 1 \bmod n$.*

Proof. Since a is coprime with n, it has some finite order k. Then by Lagrange's theorem, k divides $\phi(n)$. This implies that $kh = \phi(n)$ for some integer h. Then

$$a^{\phi(n)} \equiv a^{kh} \equiv (a^k)^h \equiv 1^h \equiv 1 \bmod n. \qquad \square$$

3.2.1 Encryption

Bob chooses two distinct, large primes p and q, and defines $n = pq$. He generates an element e that is a unit in $\mathbb{Z}_{\phi(n)}$. Bob publishes the pair (n, e) as his public key.

Imagine Alice wants to send a message to Bob. She first asks Bob for his public information. She then converts her message into a number m in $\mathbb{Z}_n$. Alice sends Bob

$$c \equiv m^e \bmod n. \tag{3.2.1}$$

In code, this might take the following form:

```
function c = RSAencrypter(m,e,n)
    c  = powMod(m,e,n);
end
```

Notice that the actual implementation of the encryption is quite simple, while the underlying mathematics is substantially more difficult. This is quite common in modern cryptosystems; once we have mastered the underlying ideas, the execution follows quite easily.

3.2.2 Decryption

Bob receives $c \equiv m^e$ from Alice. Since Bob knows p and q, he knows $\phi(n) = (p-1)(q-1)$. Therefore, he can easily compute $d = e^{-1} \bmod \phi(n)$. To recover the plaintext, Bob raises the ciphertext c to the d.

$$c^d \equiv (m^e)^d \tag{3.2.2}$$

$$\equiv m^{ed}. \tag{3.2.3}$$

Since e and d are inverses modulo $\phi(n)$, we know that ed is multiple of $\phi(n)$ plus 1 modulo n:

$$ed \equiv 1 + h\phi(n) \bmod n.$$

Bob can use this fact together with Euler's theorem to recover the plaintext.

$$c^d \equiv m^{ed} \equiv m^{1+h\phi(n)} \equiv m(m^{h\phi(n)}) \equiv m(m^{\phi(n)})^h \equiv m(1)^h \equiv m.$$

3.2.3 Security

Eve knows Bob's public information n and e, as well as the ciphertext c. She could recover the plaintext m in one of two ways. First, she could compute d and subsequently arrive at m, just as Bob does. Second, she could directly undo what Alice has done by directly computing m from $m^e \bmod n$. As it turns out, both of these are problems are thought to be very difficult.

In order to determine $d = e^{-1} \bmod \phi(n)$, Eve would need to know $\phi(n)$. But since $n = pq$ is the product of two distinct primes, we have $\phi(n) = (p-1)(q-1)$. Currently, there's no efficient algorithm to compute $\phi(n)$ without this formula. Hence, to invert e, Eve would need to figure out p and q. This is a difficult proposition called the **integer factorization problem**. Obviously factoring $n = pq$ into its primes is easy for small examples like 21 or 33. Moreover, if $p = 2$, then you can easily determine q. But in general, for large enough primes p and q (meaning hundreds of digits, typically), factoring n is very difficult. Cryptanalysts have successfully attacked RSA when $n = 768$ bits (roughly 200 decimal digits), and the current industry standard as of 2014 is that n have at least 1024 bits (roughly 300 decimal digits).

The second piece of security comes for the so-called **RSA problem**. Currently, it is very difficult to determine an m such that given n, e, and c, we have $c \equiv m^e \bmod n$. We could also consider the RSA problem as taking the e^{th} root of c modulo n. In either formulation, the modular nature of the arithmetic takes an operation that would be fairly straightforward using real numbers and makes it difficult enough to support an industrial-grade cryptosystem.

3.3 Cryptographic Hashes

The security of many modern cryptographic techniques rests on the intractability of reversing some "easy" operation. In Diffie-Hellman, we saw that while exponentiating a primitive element g in $\mathbb{Z}_n$ is easy, taking the discrete logarithm to undo this operation is thought to be very difficult. Similarly, in RSA, we saw that while multiplying distinct primes p and q is easy, factoring $n = pq$ into its constituent parts is tough. Both the discrete logarithm problem and the integer factorization problem are examples of a **one-way function**: an operation that is easy to perform and difficult to undo. In this section, we'll see another class of one-way functions called **cryptographic hash functions**. These technologies will open a wealth of applications:

- Suppose you download a large program. Getting even one bit wrong in the downloading process can cause huge problems when you execute the program. How can you be sure that you received everything correctly?
- From a little bit more nefarious point of view, suppose that you want to download a common, free program that you've found on a third party website. How can you be sure that a malicious programmer hasn't inserted a very small piece of code into an otherwise valid program that could, for instance, log all the key strokes you make?
- Imagine that Alice issues Bob a challenge to solve a really tough problem. How can Alice convince Bob that she's actually solved the problem, too, without actually sharing the answer with Bob?
- We've seen that man-in-the-middle attacks should make us cautious of accepting someone's identity online. How can Alice convince Bob that a message she's sent actually came from her and not from a malicious middleman Mal?

Normally, we think of a mathematical function as taking an element from a specified domain as an input and producing an element of a specified range as an output. For instance, we could think of the function $f(x) = x^2$ as taking any real number as an input and producing a non-negative real number as an output. A cryptographic hash function H is similar in a way. The function H takes *any* input which can be represented by a string of bits and always produces a bit string of a particular length. For instance, the SHA-1 algorithm (which will be discussed later in more detail) *always* produces a 160-bit output:

```
>> sha1('Cryptography')

ans =

b804ec5a0d83d19d8db908572f51196505d09f98
```

Here each of the 16 possible characters in the output (`0, 1, 2,..., 9, a, b,..., f`) each represent 4 bits. This **hexadecimal notation** is a compact way of writing

large bit strings: the 160-bit output of SHA-1 can be written with just 160/4 = 40 hexadecimal characters.

We call the output of a cryptographic hash the **digest** of the input. Note that for a given hash function, the digest will always have the same length, regardless of whether the input is a single number, a single word, or an entire book.

Hash	Developer(s)	Digest size (bits)	Attack
Message-Digest 5 (MD5)	Rivest	128	Practical
Message-Digest 6 (MD6)	Rivest	≤ 512	None
Secure Hash Algorithm 1 (SHA-1)	NSA	160 bits	Practical
Secure Hash Algorithm 1 (SHA-256)	NSA	256 bits	None

Table 3.1: Popular cryptographic hashes and their security status as of January 2014.

3.3.1 Properties

Just as with the discrete logarithm problem and the integer factorization problem, we would like for a cryptographic hash function to be easy to compute but difficult to "undo." We can make this concept a little more rigorous: given a hash digest value h produced by a hash function H, we would like it to be difficult to determine a message m such that $H(m) = h$. Said yet another way, we would like any hash function H to be a one-way function. In the hash function literature, this property is also known as **pre-image resistance**. Using this language, the **pre-image** is the message m and the **image** is the digest value h. So the pre-image property requires that it be difficult to find a pre-image that matches a given image.

Hash functions go a step beyond one-way functions in that they also require that it be difficult to find two different inputs that lead to the same output. More formally, a good hash function H should make it difficult to find distinct m_1 and m_2 such that $H(m_1) = H(m_2)$. We call such an occurrence a **collision**, and this property is generally known as **collision resistance**. Notice that since every bit string of length 200 (for instance) has a 160-bit SHA-1 digest, it must be the case that many inputs leads to the same digest. After all there are 2^{200} possible inputs and only 2^{160} possible outputs, so some inputs must lead to the same output. Said another way, collisions *must* happen! Collision resistance just implies that it's difficult for us to find them.

Let's see a consequence of the advantageous properties of good cryptographic hashes. Consider the following two digests generated by nearly identical messages:

```
>> sha1('Cryptography')
```

```
ans =

b804ec5a0d83d19d8db908572f51196505d09f98

>> sha1('Cryptography!')

ans =

47e8cd97112fdf810d49f802bdf17cd219e8dbab
```

While the inputs are almost identical, the outputs are incredibly different! This is a reflection of the pre-image resistance property. If $m_1 \approx m_2$ (meaning that the two input bits strings were almost identical) implied that $H(m_1) \approx H(m_2)$, we might be able to massage the inputs so that the outputs matched, resulting in a collision. This method of generating collisions fails if similar inputs are mapped to very different outputs.

3.3.2 Applications

Suppose a software development company deploys a new version of a popular program. In order to reach the largest number of end users, the company decides to post the software not just on its own website, but on third-party sites, as well. But what's stopping some nefarious agent from adding some sort of virus to the company's legitimate installer and posting this malware-ladened version for users to download? Technically speaking, absolutely nothing. We can use the concept of cryptographic hashes to give end users a method for verifying that the code they downloaded is the same (with very high probability) as the code the company released.

After finalizing the code that they will ship to consumers, the developers compute the digest of the *entire* program using some strong cryptographic hash. They post the digest h on their website. After downloading the code, an end user can compute the hash h' of the entirety of the downloaded code. If $h = h'$, then the end user can be very confident that the code they obtained is exactly the same as the code that the company released. Let's unpack this claim in a little more detail.

Now imagine that some malicious developer is trying to insert a virus into the company's code. The hacker knows the true hash h. If the hacker changes the code, then there is a very high probability that the hash will change as well. This would give the attacker away! In order for the changes to remain undetected, the hacker would have to find a collision, namely an edited version of the legitimate program whose hash was identical to h. A good hash will have strong collision resistance, making finding such collision incredibly difficult. Moreover, if the hash has strong pre-image resistance, it would be difficult for the attacker to even *find* an input which would lead to the digest h, let alone another input which contained the virus the attacker is trying to spread.

For another example, imagine that Alice wants to challenge Bob to solve a

tough problem with a unique solution. Before he starts such a tough task, he wants to have some sort of proof that Alice herself has solved the problem. The issue here is that if Alice sends the answer itself to Bob, then there's no reason for Bob to solve the problem. Let's imagine that Alice comes up with an answer a to the problem. She can send $H(a)$, where H is some hash function that Alice and Bob have agreed upon. Notice that Bob cannot recover a from $H(a)$, because hashes are one-way functions. When Bob solves his problem, he can compute the hash of the solution he's found. If the hash he computes matches the hash Alice sent, then Alice had very likely solved the problem before she sent the hash to Bob. After all, it's much more likely that Alice actually solved the problem than it is that she found a collision.

3.3.3 Cryptanalysis

To get an idea of how hard it is to find collisions for a given hash function, let's think about a seemingly unrelated problem: how likely is it that in a classroom full of people, at least one pair of people will have the same birthday? These problems aren't so unrelated, because we can think of a birthday as a sort of hash which accepts a person as an input and produces a number from 1 to 365 as its output, namely the day of the year in which the person was born.

For simplicity, let's assume that birthdays are uniformly distributed across all days of the year. This is certainly not true, but as we'll see, this assumption provides a upper bound on the difficulty of finding a pair of people with the same birthday; if we introduce the actual lumpiness into birthday distribution, finding a pair of people that share the same birthday will only get easier. Let's also assume that each person's birthday is independent from every other.

Imagine there are n people in the class, ordered 1 through n. If there are more than 365 people in the class, then at least one pair has the same birthday. So let's assume that $n \leq 365$. Person 1 has some birthday. The probability that person 2 has a different birthday than person 1 is

$$p_1 = \left(\frac{364}{365}\right).$$

There are still 365 days to choose from, but only 364 of them lead to person 2 having a different birthday than person 1. The probability that person 1, person 2, and person 3 all have different birthdays is

$$p_2 = \left(\frac{364}{365}\right)\left(\frac{363}{365}\right).$$

There are still only 364 valid choices that result in person 2 having a different birthday than person 1. When we arrive at person 3, there are only 363 choices, since both person 1 and person 2's birthdays have been removed from the pool. Notice that we can simply multiply the two probabilities together because we're assuming that each person's birthday is independently chosen from every other.

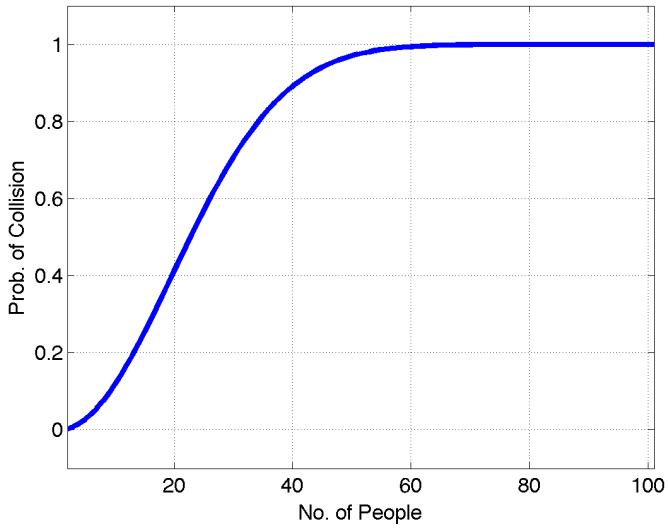

Figure 3.1: Probability that at least one pair of people share the same birthday as a function of the number of people in the group. Notice in order to have a 50% chance of having a collision, the group size must be roughly 23.

We can continue this chain of logic down to the n^{th} person. Here, the probability of all n people having different birthdays is

$$p_n = \left(\frac{364}{365}\right)\left(\frac{363}{365}\right)\cdots\left(\frac{365-n+1}{365}\right).$$

This implies that the probability that there is a collision, that is, the probability that at least one pair of people have the same birthday is

$$c(n) = 1 - \left(\frac{364}{365}\right)\left(\frac{363}{365}\right)\cdots\left(\frac{365-n+1}{365}\right). \tag{3.3.1}$$

We can see a plot of this collision probability as a function of n in Figure 3.1. There are several key features to note. First, as the size of the group increases towards $n = 365$, the probability of collision goes to 1. Second, there is a rather steep increase in collision probability from class size $n \approx 5$ to $n \approx 30$ which then flattens off as we add more people to the mix. Finally, to achieve a 50% collision probability we need approximately 23 people in the class. We'll see that this 50% mark is a convenient way to compare across different hashes and attacks.

We can extend this same methodology to the case in which a hash function has more than 365 possible outputs. Imagine that a hash function H has $|H|$ possible outputs. For instance, for a hash function H that always produces 128 bits, we have $|H| = 2^{128}$.

Hash output size (bits)	Birthday bound for 50% collision chance
64	5.1540×10^9
128	2.2136×10^{19}
256	4.0834×10^{38}
512	1.3895×10^{77}

Table 3.2: Birthday bound as hash size increases. Notice that when the hash output size is doubled, the birthday bound grows by roughly a power of 2.

A careful analysis would show that in order for an attacker to have a 50% chance of finding a collision, the attacker would need to test roughly $1.2\sqrt{|H|}$ different inputs. This quantity is known as the **birthday bound**, and correspondingly, this sort of assault on cryptographic hashes is known as the **birthday attack**. In the case of the actual birthday hash, we have

$$1.2\sqrt{|H|} = 1.2\sqrt{365} \approx 22.9.$$

This is in good agreement with our qualitative prediction from Figure 3.1.

Table 3.2 shows the birthday bound for several popular hash lengths. As the hash output size (measured in bits) is doubled, the birthday bound grows by roughly a power of 2. For instance, the birthday bound for a hash output of size 64 is "only"

$$1.2\sqrt{2^{64}} \approx 5.154 \times 10^9. \tag{3.3.2}$$

In other words, an attacker would need only check roughly 5 billion inputs in order to find a collision. While this may sound like a lot, in the age of modern computation, 5 billion is a drop in the bucket. For this reason, cryptographic hashes with such small outputs are considered insecure. Even some 128 bit hashes (for which the birthday bound is roughly 2.2×10^{19}) have been successfully attacked; see for instance MD5 from Table 3.1. The newest generation of widely accepted hash functions use at least 256 bits, for example, SHA256 and MD6 from Table 3.1.

3.4 Digital Signatures

Imagine that Alice wants not only to send Bob a message, but to convince Bob that no one has tampered with her message in transit. This will require us to combine our knowledge of encryption technologies and cryptographic hashes.

3.4.1 RSA-based Signature Scheme

Alice wants to send a secure message to Bob. She chooses primes p_a and q_a and a unit e_a in $\mathbb{Z}_{\phi(p_a q_a)}$ according to the RSA scheme. She releases $n_a = p_a q_a$ and e_a as her public keys. She retains p_a, q_a and e_a^{-1} privately. Similarly, Bob chooses p_b and q_b and a unit e_b in $\mathbb{Z}_{\phi(p_b q_b)}$, and releases $n_b = p_b q_b$ and e_b publicly. Alice and Bob publicly agree on a hash function H.

Imagine Alice wants to send a message m to Bob. She first computes the messages digest, $h = H(m)$. She then signs this digest with her *private* key:

$$s \equiv h^{e_a^{-1}} \bmod n_a \tag{3.4.1}$$

Here, we consider s to be Alice's **signature** on the message m. Alice encrypts m as usual using Bob's public key:

$$c \equiv m^{e_b} \bmod n_b. \tag{3.4.2}$$

She transmits both the ciphertext c and the signature s to Bob.

In order to verify Alice's message, Bob performs the following comparison:

$$H(c^{e_b^{-1}}) \bmod n_b = s^{e_a} \bmod n_a. \tag{3.4.3}$$

If this equation holds, then Bob can be very confident that Alice sent the message. If the equation fails to hold, Bob can be very certain that something has gone wrong.

To see why this works, let's first assume that Eve did not interfere. Then

$$c^{e_b^{-1}} \equiv m^{e_b e_b^{-1}} \equiv m \bmod n_b, \tag{3.4.4}$$

in just the same way as traditional RSA-decryption. Therefore, if nothing nefarious has occurred, Bob computes $H(c^{e_b^{-1}}) \equiv H(m) \equiv h \bmod n_b$.

Again assuming that Eve has not interfered with the transmission, the right side of 3.4.3 gives

$$s^{e_a} \equiv h^{e_a e_a^{-1}} \equiv h \bmod n_a. \tag{3.4.5}$$

Hence, if Eve has not interfered, then Bob computes the hash digest h in two different ways, and so Equation 3.4.3 holds.

Since the message Alice sends is encrypted using only Bob's public key, Eve could intercept the ciphertext c and replace it with her own message m' with associated ciphertext $c' \equiv (m')^{e_b} \bmod n_b$. From Bob's perspective, this would look totally normal. Issues start to arise when Bob examines the hash and

the signature. Notice that $H(m') \neq H(m)$, unless Eve has found a way to generate collisions in the hash function H. This implies that Eve would need to change the signature Alice sent, as well. Imagine Eve is trying to come up with a signature s' that matches her new message m' with digest $h' = H(m')$. Eve can't simply set $s' = (h')^{e_a^{-1}}$ as Alice would've done, because e_a^{-1} is Alice's *private* key. So Eve will have to compute a new signature s' from scratch. She knows Bob will raise the signature she provides to Alice's *public* key, so, in essence Eve needs to find s' such that

$$(s')^{e_a} \equiv H(m') \bmod n_a. \tag{3.4.6}$$

This is the familiar RSA problem from Section 3.2. We can conclude with reasonable certainty that Eve could not successfully forge a signature.

Example 3.4.1. *Let's imagine that Alice wants to send a signed version of the message $m = 6283$ to Bob using the RSA-based scheme. Imagine Alice has chosen $p_a = 839$ and $q_b = 557$, and published $n_a = p_a q_a = 467323$ and $e_a = 11$. Furthermore, suppose Bob has chosen $p_b = 73$ and $qb = 347$, and published $n_b = 25331$ and $e_b = 13$. The hash function* `sha0` *gives a digest of $h = 12473$. Alice sends Bob ciphertext $c \equiv m^{e_b} \bmod n_b$. We can compute this in Matlab:*

```
>> c = powMod(6283, 13, 25331)

c =

        7491
```

Alice also sends Bob the signature $s \equiv h^{e_a^{-1}} \bmod n_a$. Again, we can compute this quantity in Matlab:

```
>> eaInv = modInv(ea, (pa-1)*(qa-1));
>> s = powMod(h, eaInv, na)

s =

       17101
```

To verify Alice's signature, Bob first deciphers the ciphertext to recover m:

```
>> ebInv = modInv(eb, (pb-1)*(qb-1));
>> m = powMod(c, ebInv, nb)

m =

        6283
```

He then compares the `sha0` *hash digest of* `m`,

```
>> sha0(6283)

ans =

       12473
```

to Alice's signature raised to the power of her public key,

```
>> powMod(s, ea, na)

ans =

       12473
```

Since the two values match, Bob can be very sure that Alice sent the message.

There are two main components that make this and other signature schemes successful. The first is that the digest h is integral to the verification procedure. This makes it impossible for Eve to change only the message; she will have to change the signature as well. Second, the signature depends on Alice's *private* information in a way that makes it impossible for Eve to replicate. Together, these allow Bob to be confident after verification that Alice is who she says she is.

3.4.2 DLP-based Signature Scheme

Another signature method is based on the discrete logarithm problem. This method is commonly known as the **ElGamal signature scheme**, after its inventor Taher ElGamal.

Alice and Bob agree on a prime number p and a primitive element g in $\mathbb{Z}_p$. Alice chooses an element a in $\mathbb{Z}_p$ and keeps this information private. She releases $A \equiv g^a \bmod p$ publicly.

Imagine Alice wants to send a message m to Bob. She first chooses a nonzero number k in $\mathbb{Z}_{p-1}$. We'll see later that it's very important that Alice choose a new k each time she wants to send a message to Bob. If she doesn't, Eve will be able to easily forge Alice's signature. Once Alice has chosen k, she computes

$$r \equiv g^k \bmod p. \tag{3.4.7}$$

Alice next computes the messages digest, $h = H(m)$. She then signs the message digest with her private key a:

$$s \equiv (h - ar)k^{-1} \bmod (p-1) \tag{3.4.8}$$

Notice that here k^{-1} is the multiplicative of k in $\mathbb{Z}_{p-1}$. Alice transmits both r and s, along with an encrypted copy of the message to Bob.

To verify Alice's transmission, Bob first decrypts the ciphertext, recovers the plaintext, and computes its hash h. He then performs the comparison

$$g^h \equiv r^s A^r \bmod p, \tag{3.4.9}$$

where A is Alice's public key. If the two sides of the equivalence match, Bob can be very confident that Alice is actually the person who sent the message. If the two sides of the equivalence differ, Bob can be very confident that something has gone wrong.

To verify that Bob's comparison actually works, first imagine that Eve has not interfered with the transmission. We could rearrange Equation 3.4.8 to read

$$h \equiv sk + ar \bmod (p-1) \tag{3.4.10}$$
$$\Rightarrow h = \ell(p-1) + sk + ar. \tag{3.4.11}$$

Then raising g to the power h gives

$$g^h \equiv g^{\ell(p-1)+sk+ar} \bmod p \tag{3.4.12}$$
$$\equiv g^{\ell(p-1)} g^{sk} g^{ar} \bmod p \tag{3.4.13}$$
$$\equiv r^s A^r \bmod p, \tag{3.4.14}$$

where $g^{\ell(p-1)} \equiv 1 \bmod p$ through Euler's theorem.

Now imagine that Eve tries to interfere with the transmission by replacing Alice's message m with her own message m'. Unless Eve has worked out a way to force collisions in H, the hash of her new message m' will not match the hash of the original message m. Therefore, Eve must replace Alice's signature r and s with her own r' and s'. She can't simply set s' to be equal to s. To see why, let's take another look at Equation 3.4.8. We see that both the digest h and Alice's *private* key a are used in the computation of s. Moreover, these quantities are "disguised" via multiplication by k^{-1}. Recall that Alice chose k privately, and chooses a different k for each message she wants to send to Bob.

Example 3.4.2. *Suppose Alice and Bob have agreed on $p = 937$ and $g = 7$. We can verify that g is primitive in $\mathbb{Z}_{937}$:*

```
>> ord(7, 937)

ans =

   936
```

Imagine Alice wants to send a signed version of the message $m = 6283$ to Bob using the DLP-based scheme. Alice chooses a private key, say $a = 12$, and releases $A = g^a \bmod p \equiv 35 \bmod p$ as her public key. The `sha0` *message digest is*

```
>> h = sha0(6283)

h =

       12473
```

Alice chooses a random unit k in $\mathbb{Z}_{p-1}$, and a private key. We can verify that $k = 11$ is such an element:

```
>> gcd(11, p-1)

ans =

     1
```

Alice then computes $r \equiv g^k \bmod p$ and $s \equiv (h - ar)k^{-1} \bmod (p-1)$.

```
>> r = powMod(g, k, p)

r =

     5

>> s = mod((h - a*r)*modInv(k, p-1), p-1)

s =

   703
```

Notice that the inverse of k is computed in $\mathbb{Z}_{p-1}$ not $\mathbb{Z}_p$.

Upon receiving the ciphertext, Bob first recovers the message m, and then computes its hash $h = H(m)$. He then compares

```
>> powMod(g, h, p)

ans =

   769
```

to the alternate expression

```
>> mod(powMod(r, s, p)*powMod(A, r, p), p)

ans =

   769
```

Since the two values match, Bob can be very sure that Alice did indeed send the message she claimed.

We've claimed that it's important for Alice to choose a new value of k each time she sends a message to Bob. Let's see what happens if she fails to choose a new k each time. Suppose that Eve happens to know two hashes h_1 and h_2 and

the related signatures s_1 and s_2. The way in which Alice forms her signatures is public knowledge, so Eve knows

$$s_1 k + ar = h_1 \bmod (p-1) \tag{3.4.15}$$
$$s_2 k + ar = h_2 \bmod (p-1). \tag{3.4.16}$$

She can subtract the second equation from the first to obtain

$$(s_1 - s_2)k = (h_1 - h_2) \bmod (p-1). \tag{3.4.17}$$

If $(s_1 - s_2)$ is invertible in $\mathbb{Z}_{p-1}$, then Eve can solve for k. This is bad enough, but Eve can further exploit her knowledge. With k, Eve can compute r. Substituting k and r into $s_1 k + ar = h_1 \bmod (p-1)$, Eve can solve for Alice's private key a. This is *incredibly* bad! After all, once Eve knows Alice's private key, she can start signing things just as if she were Alice! For the rest of the cryptographic world, there is no functional equivalent between Eve and Alice at all!

This may seem all rather esoteric, but this attack has played out in practice. The **Digital Signature Algorithm** is very similar to the ElGamal signature scheme and has been widely used in industry. In 2010, the PlayStation3 console used DSA to sign digital content, so that users could be sure that the software they were installing on their expensive consoles came from Sony, not from some malicious developer. But Sony failed to choose a new k for each message they sent to users. A group of hackers known as `fail0verflow` exploited this fact in order to gain Sony's private keys. The group was then able to impersonate Sony on any PS3 box. This tale should reinforce the concept that a cryptosystem is only as good as its implementation; if we fail to implement things properly, even a system that appears strong on paper can fail in catastrophic ways.

3.5 Zero-Knowledge Proofs

We introduce two new characters: Peggy the prover, and Victor the verifier. Imagine that Peggy has told Victor that she's invented a machine that can see through walls. Peggy is understandably protective about her technology, and does't want Victor to see or inspect the machine lest he somehow figure out the secret of how it works. At the same time, Peggy would like to convince Victor that the machine does in fact do what she claims. How can Peggy prove to Victor in a verifiable way that she has this secret information?

A demonstration of the technology seems appropriate. Victor sets up a wall behind which he places either one or two boxes. Let's assume he decides by flipping a fair coin. He'll then ask Peggy to tell him whether she sees one or two boxes behind the wall. If Peggy's machine works, she'll have no problem answering correctly. If Peggy's been bluffing, however, she'll have to guess. In this case, she'll have a 50% chance of fooling Victor. Obviously, Victor doesn't want to be tricked. Fortunately, he has a solution. He flips his fair coin again, places either one or two boxes behind the wall, depending on the coin toss's outcome, and asks Peggy to answer again. Again, Peggy has a 50% chance to guess correctly if her machine doesn't work. But since Victor's fair coin flips are independent of one another, Peggy only has a $(1/2)^2$ chance of getting *both* guesses correct.

Victor performs k of these rounds. If Peggy's machine works, she has no problem responding correctly to all k challenges. If her machine does not work, she will guess correctly all k times with probability $(1/2)^k$. Notice that as we'd expect, the probability that Peggy can fool Victor decreases as k increases. In fact, it decreases *exponentially* fast. Hence, by choosing large enough k, Victor can be as sure as he'd like to be that Peggy is telling the truth.

This toy example gives us some good insights into the advantageous properties of a **zero-knowledge proof**. There are three key properties that every such demonstration should have:

- Completeness: if Peggy is telling the truth, she can convince Victor
- Soundness: if Peggy is not telling the truth, she will fail to fool Victor with very high probablity
- Zero-knowledge: Peggy does not give any information about how she knows what she knows to Victor in the act of convincing him

These properties may be a little more involved than they might first appear. For instance, soundness requires that if Peggy is lying, she fails to fool Victor with *high* probability. In the example above, Peggy fails with probability $1 - 2^{-k}$. The surprising thing might be that we do not require that Peggy *always* fail to fool Victor if she is in fact lying; it's enough for a zero-knowledge proof that the she fail *most* of the time. Typically, we'd like to allow Victor to set his level of confidence, as was the case in the example above.

The zero-knowledge condition also has some interesting implications. In the example above, we stipulated that not only could Victor not examine the inner workings of Peggy's machine, but he was not even allowed to *see* it. After all, even seeing the size or shape or rough layout of the machine may have given Victor some information as to how Peggy was accomplishing her feat. This is the true sense of zero-knowledge: not only is the secret itself not revealed to Victor, but no information is revealed that could even get Victor *closer* to knowing what Peggy knows. While it's often easy to see that the secret itself has not been revealed, it's often more difficult to prove concretely that Victor can't somehow use the information provided to him to get closer to the secret. We'll see several examples of this over the course of this section.

3.5.1 Schnorr Authentication

Peggy wants to prove to Victor she knows the discrete logarithm of an element $y \equiv g^x \bmod p$ in $\mathbb{Z}_p$ without revealing the logarithm x to Victor. We could imagine, for instance that x is Peggy's private key, and y is Peggy's public key. Then Peggy essentially would like to convince Victor that she knows the private key x, *i.e.*, that she is in fact Peggy. This process of convincing someone you are who you say you are is known as **authentication**. The following authentication protocol which relies on the discrete logarithm problem was developed by Schnorr in 1991.

Peggy and Victor agree to perform k rounds of the following algorithm:

- Peggy generates random integer r and sends $C \equiv g^r \bmod p$ to Victor.
- After receiving C, Victor generates a random bit b. If $b = 0$, he tells Peggy to reveal r. If $b = 1$, he tells Peggy to reveal $w = x + r \bmod (p-1)$.
- Once he receives the revealed value from Peggy, he verifies that $g^r \equiv C \bmod p$ if his random bit was $b = 0$, and he verifies that $Cy \equiv g^w \bmod p$ if his random bit was $b = 1$.

If Victor's verification works for all k rounds, Victor concludes that Peggy probably knows the value of x. If Victor's verification *ever* fails, he concludes that Peggy is lying.

Example 3.5.1. *Imagine Peggy and Victor have agreed on* $p = 997$ *and* $g = 111$. *Peggy wants to prove to Victor that she knows that* $322 \equiv 111^{34} \bmod 997$ *without revealing the exponent* $x = 34$ *that would lead to* $y = 322$.

Peggy begins the round by choosing a random r *in* $\mathbb{Z}_{997}$, *for instance* $r = 968$. *She then sends* $C \equiv g^r \bmod p$ *to Victor:*

```
>> C = powMod(g, r, p)

C =

   270
```

Victor randomly generates a bit $b = 1$. If $b = 0$, he tells Peggy to reveal r, and if $b = 1$, he tells Peggy to reveal $x + r \bmod p$. Given his choice of $b = 1$, Peggy sends Victor $w \equiv x + r \bmod p$:

```
>> >> w = mod(x + r, p - 1)

w =

     6
```

Victor verifies that $g^w \equiv Cy \bmod p$:

```
>> mod(C*y, p)

ans =

   201

>> powMod(g, w, p)

ans =

   201
```

Since the two quantities match, Victor concludes that Peggy probably knows x. If instead Victor had chosen $b = 0$, Peggy would've sent the value of r. Victor would've then verified that $g^r \equiv y \bmod p$.

Completeness: If Peggy knows x, she can correctly respond to either of Victor's requests. If Victor's random bit is $b = 0$, then Peggy must simply send r; she can accomplish this regardless of whether she knows x. If Victor's random bit is $c = 1$, then Peggy must send $w \equiv x + r \bmod (p - 1)$. Notice that since $w \equiv x + r \mod (p - 1)$, we have $w = \ell(p - 1) + x + r$ for some whole number ℓ, and so

$$\begin{aligned} g^w &\equiv g^{\ell(p-1)+x+r} && (3.5.1)\\ &\equiv g^{\ell(p-1)} g^x g^r && (3.5.2)\\ &\equiv g^x g^r && (3.5.3)\\ &\equiv Cy \bmod p && (3.5.4) \end{aligned}$$

We move from the second to third equivalence via Euler's theorem: $g^{p-1} \equiv g^{\phi(p)} \equiv 1 \bmod p$.

Soundness: Suppose Peggy does not know x. If Peggy knew that Victor would reveal $b = 0$, she could simply choose any r and send $C \equiv g^r \bmod p$ as her commitment. However, if she sent this value of C and Victor instead chose $b = 1$, she would be in trouble, because she would need to a send a value z such

that $g^z \equiv Cy \bmod p$. But by the assumed difficulty of the discrete logarithm problem, Peggy would not be able to find such a z.

On the other hand, if Peggy knew that Victor would reveal $b = 1$, she could choose a random r and send $C \equiv g^r y^{-1} \bmod p$ as her commitment. This implies that $g^r \equiv Cy \bmod p$, and so Victor's verification would work perfectly if he chose $b = 1$. However, if Peggy was wrong, and Victor chose $b = 0$, the value of r she would send would fail Victor's verification $g^r \equiv C \bmod p$.

Since in reality Peggy does not know which value of b Victor will pick, we should expect that Peggy would fail Victor's challenge 50% of the time if she did not in fact know the value of x. The probability that Peggy could fool Victor in k sequential rounds is $(1/2)^k$. By making k large, we can satisfy the soundness criterion.

Zero-knowledge: In each round, Peggy chooses a new value of r and either reveals r or $x + r \bmod p$, depending on Victor's request. If Peggy choice of r is random and unrelated to her past choices of r, then Victor gains no knowledge about the secret value of x, provided we assume that he cannot take the discrete logarithm of $C \equiv g^r \bmod p$.

Notice we could've written the algorithm in a slightly more compact form:

- Peggy generates random integer r and sends $C \equiv g^r \bmod p$ to Victor.
- After receiving C, Victor generates a random bit b and sends it to Peggy. Peggy returns $w \equiv r + bx \bmod (p - 1)$.
- Once he receives the revealed value from Peggy, he verifies that $g^w \equiv Cy^b \bmod p$.

This makes the proofs of completeness, soundness, and zero-knowledge a little more compact at the expense of being a little less intuitive. Understanding this type of notation is important for fully appreciating another of the common applications of zero-knowledge proofs, the Feige-Fiat-Shamir authentication scheme.

3.5.2 Feige-Fiat-Shamir Authentication

Many commonly used cryptographic systems rely on a **trusted third party**, also known as a TTP. A TTP can perform all sorts of interesting and useful functionalities, including keeping records and certificates of users' public keys. In 1988, Feige, Fiat, and Shamir developed a zero-knowledge method in which a TTP can help facilitate positive identification of one user by another.

Suppose Victor wants to verify Peggy's identity through a TTP. The TTP first generates an RSA modulus $n = pq$ for large, distinct primes p and q. The TTP also creates a collection of private keys $s_1, s_2, \ldots, s_k$, each satisfying $\gcd(s_i, n) = 1$ for Peggy. It keeps the associated public keys $t_i \equiv s_i^2 \bmod n$ for $i = 1, 2, \ldots, k$ on file.

Peggy would like to convince Victor that she knows the private keys $s_1, s_2, \ldots, s_k$ without revealing these secret keys to him. Victor begins the transaction by contacting the TTP for Peggy's public keys. Once Victor has the public keys, Peggy and Victor agree to complete ℓ rounds to the following algorithm:

- Peggy choose a random $m \in \mathbb{Z}_n$. She sends $w = m^2 \bmod n$ to Victor.
- Victor chooses a random bits $c_1, c_2, \ldots, c_k$ and sends them to Peggy.
- Peggy computes $r \equiv m s_1^{c_1} s_2^{c_2} \ldots s_k^{c_k} \bmod n$ and sends it to Victor.
- Victor computes r^2. If $r^2 \equiv w t_1^{c_1} t_2^{c_2} \ldots t_k^{c_k}$, then Victor accepts Peggy's demonstration.

Example 3.5.2. *Imagine that the TTP has chosen* $p = 911$ *and* $q = 997$ *for its primes, and distributed the secrets* $s_1 = 17$ *and* $s_2 = 71$ *to Peggy. It has retained the public keys* $t_1 \equiv s_1^2 \equiv 289 \bmod n$ *and* $t_2 \equiv s_2^2 \equiv 5041 \bmod n$ *on file.*

In the first round, suppose Peggy choose $m = 869363$*. She sends*

```
>> w = powMod(m, 2, n)

w =

      348394
```

to Victor. Victor in turn generates random bits $c_1 = 1$ *and* $c_2 = 0$*, which he sends to Peggy. Peggy then computes*

$$r \equiv m s_1^{c_1} s_2^{c_2} \bmod n \tag{3.5.5}$$

In Matlab,

```
>> r = mod(m * s_1^1 * s_2^0, n)

r =

      246899
```

She sends this value of r *to Victor. Victor then verifies that* $r^2 \equiv w t_1^{c_1} t_2^{c_2} \bmod n$.

```
>> powMod(r, 2, n)

ans =

      776496

>> mod(w*t1^1*t2^0, n)

ans =

      776496
```

Since the two values match, Victor concludes that Peggy probably has access to the private keys s_1 *and* s_2.

Completeness: Note that

$$r^2 \equiv (ms_1^{c_1} s_2^{c_2} \dots s_k^{c_k})(ms_1^{c_1} s_2^{c_2} \dots s_k^{c_k}) \bmod n \tag{3.5.6}$$

$$\equiv m^2 \left(s_1^2\right)^{c_1} \left(s_2^2\right)^{c_2} \dots \left(s_k^2\right)^{c_k} \bmod n \tag{3.5.7}$$

$$\equiv w t_1^{c_1} t_2^{c_2} \dots t_k^{c_k} \bmod n. \tag{3.5.8}$$

Therefore if Peggy has access to the secrets $s_1, s_2, \dots, s_k$, Victor's verification will succeed.

Soundness: Suppose Peggy does not know the secrets $s_1, s_2, \dots, s_k$. If she could somehow correctly guess which of Victor's random bits would be 1, she could choose a value of m such that Victor's verification would succeed. However, since Victor only chooses his random bits after receiving $w \equiv m^2 \bmod n$ from Peggy, Peggy cannot simply choose an m to suit Victor's revealed random bits; her choice of m is already encoded in w. Therefore, the best Peggy could do would be to guess which of the k bits will be 1 and which will be 0. There is a 2^{-k} chance she guesses correctly. Over ℓ rounds, there is a $2^{-k\ell}$ chance that Peggy will fool Victor into believing that she has access to the secret numbers. If there are $k = 6$ secret numbers and $\ell = 5$ rounds, there roughly a 1 in 1,000,000,000 chance that Peggy fools Victor.

Zero-knowledge: Peggy reveals $w \equiv m^2 \bmod n$ and $r \equiv ms_1^{c_1} \dots s_k^{c_k}$ to Victor. Note that Victor cannot determine m from w due to the difficulty of the RSA problem. This value of m also masks the secrets hidden in r. So Victor cannot determine the secrets from the information passed in a single round. Moreover, since Peggy chooses a new value of m each round, Victor cannot combine the information from multiple rounds in order to determine the secrets $s_1, \dots, s_k$.

3.6 Ethics in Cryptography

3.6.1 Obligations to Customers

Companies must store users' data to provide basic services. To what extent is a company ethically required to provide strong security for their users' sensitive data?

Selected Readings

1. Goodin, Dan. "How an Epic Blunder by Adobe Could Strengthen Hand of Password Crackers" *Ars Technica*. 1 Nov. 2013, visited 15 April 2014. Clickable link.

2. Henschke, Adam. "Encryption Ethics: Are Email Providers Responsible for Privacy?" *The Conversation*. 27 Nov. 2013. 15 April 2014. Clickable link.

3. Schuman, Evan. "Evan Schuman: Starbucks Caught Storing Mobile Passwords in Clear Text" *Computerworld*. 15 Jan. 2014, visited 15 Apr. 2014. Clickable link.

4. Silver, Joe. "Lavabit Held in Contempt of Court for Printing Crypto Key in Tiny Font [Updated]" *Ars Technica*. 16 Apr. 2014, visited 8 May 2014. Clickable link.

Discussion Questions

- Can the average user make truly informed decisions about whether a given cryptographic solution is sound?

- If the average user cannot make informed decisions about digital security, to what extent is a private company obligated to protect them?

- What effect does the failure of one website have on the rest of the ecosystem?

- If a single failure has a net negative impact on the ecosystem, does a private company have an obligation to its peers to implement strong security?

- Do repeated, high-profile hacks erode public confidence in cryptography in general?

- There is often a trade-off between security and expense. How can a company go about deciding where along this security-expense curve to operate?

3.6.2 Utility of Hacking

One could argue that allowing the "good guys" to have unfettered access to data makes us all safer, *e.g.*, by thwarting terrorism. One could also argue that such access would allow tyrannical governments to more effective persecute their citizens. To what extent should sovereign powers be allowed access to their citizens' digital data and communications ethically?

Selected Readings

1. Musil, Steven. "Saudi Arabia Announces BlackBerry Ban" *CNET*. 3 Aug. 2010, visited 15 Apr. 2014. Clickable link.

2. Chung, Emily. "Wiretap Laws Apply to Text Messages, Court Rules" *CBC News*. 27 Mar. 2013, visited 15 April 2014. Clickable link.

3. Regalado, Antonio. "Cryptographers Have an Ethics Problem" *MIT Technology Review*. 13 Sep. 2013, visited 8 May 2014. Clickable link.

4. Simonite, Tom. "NSA Leak Leaves CryptoMath Intact but Highlights Known Workarounds" *MIT Technology Review*. 9 Sep. 2013, visited 15 Apr. 2014. Clickable link.

5. Snowden, Edward. "Statement by Edward Snowden to Human Rights Groups at Moscow's Sheremetyevo Airport" *Wikileaks*. 12 Jul. 2013, visited 24 Apr. 2014. Clickable link.

Discussion Questions

- Should citizens be allowed to use modern cryptographic tools at all?

- If private key cryptography should be legal, is it ethical for the federal government be given access by default to all private keys or other information necessary to completely decrypt encrypted data?

- Is it ethical to perform wide scale surveillance in order to detect rare events?

- If there is a trade-off between privacy and security, how should a populace go about deciding where along the privacy-security curve to operate?

- Should there be a difference in how we allow authorities to collect phone and/or written records and how we allow authorities to collect digital records?

3.6.3 What to do with a break-through

Researchers can sometimes make surprising discoveries, including finding a critical flaw in the implementation of a seemingly strong cryptosystem or inventing new mathematics that significantly decreases the difficulty of classically hard problems, *e.g.*, DLP or RSA. Given such a realization, what is the researcher ethically required to do with the information she has uncovered?

Selected Readings

1. Schneier, Bruce. "Cryptanalysis of SHA-1" *Schneier on Security*. 18 Feb. 2005, visited 8 May 2014. Clickable link.

2. Whittaker, Zack. "MD5 Password Scrambler 'No Longer Safe'" *ZDNet*. 7 Jun. 2012, visited 15 Apr. 2014. Clickable link.

3. Grubb, Ben. "Heartbleed Disclosure Timeline: Who Knew What and When" *The Sydney Morning Herald*. 15 Apr. 2014, visited 8 May 2014. Clickable link.

Discussion Questions

- Are you morally obligated to publicly release any major flaw in a cryptosystem and/or its implementation?
- Is the government morally obligated to publicly release any major flaw in a cryptosystem and/or its implementation?
- Are you morally obligated to disclose any major breakthrough to your government first, *i.e.*, before public disclosure?
- In general, is any type of preferential disclosure ethically acceptable?

Appendix A

Studio problems

A.1 Studio 1.1: Communication Systems

(1) What is Shannon's maxim?

(2) Convert the following text into its associated numeric vector:

Go, Bruins!

(3) Convert the text located in `two_cities_short.txt` into its associated vector of numeric values.

(4) Convert the following numeric characters into their associated letters:

```
[19, 7, 4, 1, 14, 18, 19, 14, 13, 19, 4, 0, 15, 0, 17, 19, 24]
```

(5) Load the data found in the file `communication_systems_numbers_1.mat` by typing `load communication_systems_numbers_1.mat` in the command line of Matlab. You should now have a variable called `plaintext` in your workspace. (You can verify that this variable is actually there by entering `whos` into the command line or by entering the variable name itself into the command line.) Convert the numeric values in `plaintext` into their associated letters.

A.2 Studio 1.2: Matlab

(1) Generate an array of whole numbers from 10 to 20. Multiply the array by 3.5, then add 7. What is the 11^{th} element of the final array?

(2) Write a function that takes an array `x` as an input and doubles every entry in `x` to form `y` as an output. Call your function `arrayDoubler`.

(3) Write a function that takes an array `x` as an input and returns the entry in the middle of the array if the array has an odd number of entries and returns the entry just to the left of the middle of the array if the array has an even number of entries. (Hint: check out the `round` function)

(4) The **factorial** of a number n is defined by $n! = n(n-1)(n-2)\cdots(2)(1)$. Write a function called `factorial` that accepts a number `n` as an input, and returns the quantity $n!$ as an output. (Hint: use a `for` loop!)

(5) Write a function that takes an input whole number `x`. If `x` is less than 10, your function should turn a matrix with 3 rows and 2 columns containing only zeros. If `x` is greater than or equal to 10, your function should return a matrix with 2 rows and 3 columns containing only ones. (Hint: check out the functions `zeros` and `ones`.)

A.3 Studio 2.1: Transposition Ciphers

(1) Encipher the plaintext "I made him some coffee" using a transposition cipher with key $k = 4$.

(2) Encipher the plaintext found in `earnest_short.txt` using a transposition cipher with key $k = 64$.

(3) Decipher the ciphertext

```
IHSFI TOMAR HKGIE ISSVM AEIA
```

assuming a transposition cipher with key $k = 3$.

(4) Decipher the ciphertext located in `ciphertext_transposition_1.txt` assuming a transposition cipher with key $k = 64$.

(5) Decipher the ciphertext located in `ciphertext_transposition_no_key_1.txt` assuming a transposition cipher with no knowledge of the key. (Hint: to convert to the pseudocode found in the text, learn about the `mod` function in Matlab.)

A.4 Studio 2.2: Caesar Ciphers

(1) Encrypt the following message using a Caesar cipher with key $k = 25$ by hand:

```
THEBA NKERO FFERS HERAB RIBE
```

(2) Encrypt the plaintext located in `text/jungle_short.txt` using a Caesar cipher with key $k = 4$ using code you have written.

(3) Decrypt the following message assuming a Caesar cipher with key $k = 3$ by hand:

```
WKHBU HDGWK HFKLO GUHQD VWRUB
```

(4) Decrypt the ciphertext located in `text/ciphertext_caesar_1.txt` assuming a Caesar cipher with key $k = 17$ using code you have written.

(5) Decrypt the ciphertext located in `text/ciphertext_caesar_no_key_1.txt` assuming Caesar cipher but no knowledge of the key k.

(6) Show that $\{1, 2, \ldots, 6\}$ is a group under multiplication modulo 7.

(7) Show that $\{1, 2, 3, 4, 5\}$ is not a group under multiplication modulo 6.

(8) Suppose we have a group G. Show that a has a unique inverse b. (Hint: suppose a has two inverses, b and c such that $b \neq c$, and see if you can get a contradiction.)

(9) Is the set of all possible fractions, commonly denoted by the symbol $\mathbb{Q}$, a group under multiplication?

A.5 Studio 2.3: Affine Ciphers

(1) Is the key pair $(k, \ell) = (3, 2)$ appropriate in affine ciphering on the English alphabet?

(2) Perform affine encryption assuming key $(k, \ell) = (5, 3)$ on the plaintext located in `text/rings.txt`.

(3) Perform affine decryption assuming key $(k, \ell) = (0, 7)$ on the ciphertext located in `text/ciphertext_affine_1.txt`.

(4) Perform affine decryption assuming no knowledge of the key on the ciphertext located in `text/ciphertext_affine_no_key_1.txt`

(5) Show that $(ab) \bmod n \equiv (a \bmod n)(b \bmod n)$. (Hint: write $a = q_a n + r_a$ and similarly for b.)

(6) Show that the collection of all polynomial functions $\mathcal{P}$ is a ring.

(7) Form a complete table listing all the multiplicative inverse pairs in $\mathbb{Z}_{26}$. Clearly denote all zero divisors as well.

(8) Show that every nonzero element in $\mathbb{Z}_p$ has a multiplicative inverse if p is prime.

(9) Show that while $\mathbb{Z}_n$ is never a multiplicative group, the collection of units $\mathbb{Z}_n^\times$ is always a multiplicative group.

(10) We will show that an element is a zero divisor in $\mathbb{Z}_n$ if and only if it does not have an inverse in $\mathbb{Z}_n$.

(a) Suppose that an element x is a zero divisor in $\mathbb{Z}_n$. Show that x cannot have a multiplicative inverse. (Hint: follow the general strategy found taken in the notes.)

(b) Now suppose that x does not have a multiplicative inverse. Show that x must be a zero divisor. (Hint: think about xa_1, xa_2, ..., xa_n for all the elements $a_i \in \mathbb{Z}_n$, and remember that we've assumed that there is no a_i such that $xa_i \equiv 1$.)

A.6 Studio 2.4: Polyalphabetic Ciphers

(1) Encrypt the plaintext "Cryptology" using Vigenère encryption with the key "KEY".

(2) Decrypt the ciphertext

```
DLCFM EORCB IASTF OV
```

assuming Vigenère encryption with key "KEY".

(3) Encrypt the plaintext located in the file `text/cryptography_short.txt` using the Vigenère cipher with key "UNBREAKABLE".

(4) Decrypt the ciphertext located in the file `text/ciphertext_polyalphabetic_1.txt` assuming Vigenère encryption with key "UNBREAKABLE".

(5) Encrypt the plaintext located in `text/memphis.txt` using a running key cipher with the text in `text/yankee.txt` as the key.

(6) Decrypt the ciphertext located in `text/ciphertext_polyalphabetic_2.txt` assuming encryption with a running key cipher with the text in `text/yankee.txt` as the key.

(7) Decrypt the ciphertext located in `text/ciphertext_polyalphabetic_no_key_1.txt` assuming Vigenère encryption with no knowledge of the key.

A.7 Studio 3.1: Diffie-Hellman

(1) Is 2 primitive modulo 13?

(2) Is 3 primitive modulo 13?

(3) Assume that Alice and Bob perform Diffie-Hellman key exchange using prime $p = 61$ and primitive element $g = 10$. Imagine that Alice's private key is $a = 7$ and Bob's is $b = 50$. Compute the public message Alice sends Bob, the public message Bob sends Alice, and the shared secret number at the end of the exchange.

(4) Assume that Alice and Bob perform Diffie-Hellman key exchange using prime $p = 997$ and primitive element $g = 11$. Imagine Alice's private key is $a = 812$ and Bob's is $b = 903$. Compute the public message Alice sends Bob, the public message Bob sends Alice, and the shared secret number at the end of the exchange.

(5) Find two widely accepted technologies, current or formerly in use, that use Diffie-Hellman key exchange.

A.8 Studio 3.2: RSA

(1) Show that every zero divisor has infinite order. (Hint: suppose it has order k and try to find a contradiction.)

(2) Show that every unit in $\mathbb{Z}_n$ has order less than or equal to $\phi(n)$. (Hint: suppose a unit x has order greater than $\phi(n)$, and consider the powers $x^1, x^2, \ldots, x^{\phi(n)}$.)

(3) Show that $\phi(p^2) = p^2 - p$ if p is prime.

(4) Prove the following claim: if an element x has order $\phi(n)$ in $\mathbb{Z}_n$, then x is a primitive element in $\mathbb{Z}_n$.

(5) Imagine an RSA cryptosystem is created with primes $p = 991$ and $q = 113$.

(a) Imagine Alice chooses $e = 50$. Is this value appropriate? Why or why not?

(b) Imagine Alice now chooses $e = 53$. Verify that this choice is acceptable. What information does Alice publish?

(c) Imagine Bob wants to send the message $m = 100$. Is this value OK? Why or why not?

(d) What ciphertext does Bob send across the channel given the numbers we've decided on so far?

(6) Imagine Bob has chosen $p = 937$, $q = 683$, and $e = 383179$.

(a) Is Bob's choice of e appropriate? Why or why not?

(b) Alice sends Bob $c = 170133$. What is the associated plaintext?

A.9 Studio 3.3: Cryptographic Hashes

(1) Generating truly random bits is very difficult. In practice, we often settle for *pseudorandom bits*, meaning that the bits "look" random but are generated by a deterministic (*i.e.*, non-random) procedure. Explain how a hash function could be used to generate pseudorandom numbers.

(2) Passwords are often hashed before they are stored on a company's website.

(a) Imagine you provide your password when you try to login to a website. The web server will compare the password you provide to the hashed password it has associated with your account. Use the properties of hashes to explain why this procedure will allow you to login if and only if you've provided the correct password (with very high probability).

(b) Why wouldn't the company just compare the password you provide to a plaintext version of your password it stores on its servers?

(c) Experienced hackers know that companies store their passwords in hashed form. To speed up their attacks, hackers will compute the hashes of a bunch of common passwords (*e.g.*, "password", "1234", "asdf"). This collection of pre-computed hashes is called a *rainbow table*. (Don't believe the silly name? Check out https://www.freerainbowtables.com)Suppose a hacker manages to obtain a company's list of hashed passwords. How can a rainbow table be used to further compromise security?

(d) To mitigate the effectiveness of rainbow attacks, many websites use *salts*. A salt is a random number that is appended before your password before hashing. For a concrete example, imagine that you register at a new website. The server will generate a random number and assign this number as your account's salt. The server will then store $H([\text{salt password}])$. Explain how the use of salts makes rainbow tables far less effective, even if the hacker happens to know the salt that was applied to each password.

(3) One of the problems with using digital photography for collecting criminal evidence is the possibility of tampering and digital editing. Explain how a cryptography hash could be used to show that a photo has not been altered since it was taken. (Hint: feel free to use the time stamp of when the photo was taken.)

(4) In a silent auction, bidders write down their bids for an item and submit these bids to the auctioneer. At the end of the silent auction, the highest bid wins. Hence, there is a strong incentive to be the highest bid, but as little as possible over the runner-up.

One problem with silent auctions is that the auctioneer can collude with an accomplice by telling her the highest bid. This allows the accomplice to win the auction without overspending by any more than they must.

It would be great if each bidder had to *commit* publicly to their bid. The problem is that if they reveal their bid directly, it defeats the point of a silent auction. Devise a scheme using cryptographic hashes that allows a bidder to show the public that they have locked in their bid. Your scheme should also prevent other bidders from figuring what exactly the bid was, even if they know, for instance, that the only possible bids are increments of $10.

A.10 Studio 3.4: Digital Signatures

(1) Imagine Alice wants to send a signed version of the message `Meet at midnight` to Bob using the RSA-based signature scheme.

(a) Use `sha0` to compute the message digest h of Alice's message.

(b) If Alice chooses $p_a = 911$, $q_a = 937$ and $e_a = 11$, what signature s associated to the given message does she send Bob?

(2) Imagine now that Alice wants to send the numeric message $m = 123456789$ to Bob using an RSA-based signature scheme. Imagine that Alice has chosen $p_a = 911$, $q_a = 937$ and $e_a = 11$, and that Bob has published $n_b = 27491$ and $e_b = 13$.

(a) Use `sha0` to compute the message digest h of Alice's message.

(b) What is the ciphertext that Alice sends Bob?

(c) What is the signature that Alice sends Bob?

(3) Imagine Alice has published $n_a = 853607$ and $e_a = 11$, and Bob has chosen $p_b = 37$, $q_b = 743$, and $e_b = 13$, both using the RSA scheme. Bob receives $c = 2409$ and $s = 790832$ from someone claiming to be Alice. Should Bob believe that this message actually came from Alice?

(4) Imagine Alice has published $n_a = 853607$ and $e_a = 11$, and Bob has chosen $p_b = 37$, $q_b = 743$, and $e_b = 13$, both using the RSA scheme. Bob receives $c = 182$ and $s = 46937$ from someone claiming to be Alice. Should Bob believe that this message actually came from Alice?

(5) Imagine Alice wants to send Bob a signed version of the message `Meet at midnight` using the DLP-based signature scheme with primitive element $g = 7$ in $\mathbb{Z}_{937}$. Assume that Alice and Bob have agreed to use the `sha0` hash. Alice's private key is $a = 117$, and for this transmission, she has chosen $k = 17$. What information does Alice send Bob?

(6) Imagine now that Alice wants to send Bob a signed version of the message $m = 1234$ using the ElGamal signature scheme. Assume that Alice and Bob have agreed on $g = 2$, $p = 1117$, and `sha0` as their hashing algorithm.

(a) Show that Alice and Bob's choice of g is appropriate given their choice of p.

(b) Imagine that Alice has published $A = 501$ as her public key. If Bob receives message $m = 1234$ and signature $(r, s) = (539, 37)$ from a person claiming to be Alice, can Bob conclude that the message actually came from Alice?

(c) Imagine that Alice has published $A = 501$ as her public key. If Bob receives message $m = 5678$ and signature $(r, s) = (539, 542)$ from a person claiming to be Alice, can Bob conclude that the message actually came from Alice?

(d) Imagine that Alice has published $A = 501$ as her public key. Suppose Eve has determined that Alice is failing to choose a new value of k each time she sends a message to Bob. If Eve has collected two hash-signature pairs $(h_1, s_1) = (12473, 69)$ and $(h_2, s_2) = (3378, 446)$, what is Alice's private key a?

A.11 Studio 3.5: Zero-Knowledge Proofs

(1) A friend of yours is colorblind and so cannot tell his red socks from his green socks. In fact, he believes that they are actually the same color. You keep trying to tell him that the colors are different, that he looks like a fool with them on, and that he needs to get rid of them. Your friend thinks that you're trying to play a joke on him with all of this "different colors" business.

Fortunately, you've devised a plan to break the stalemate. You give him both a red and a green sock, and clearly tell him which is which. You tell him to randomly choose whether or not to switch the socks behind his back. He then reveals each hand to you, and you tell him which hand contains which sock. You can repeat the process over and over until your friend is sure you're telling the truth.

(a) Argue that this scheme is complete.

(b) Argue that this scheme is sound.

(c) Argue that this scheme is zero-knowledge.

(2) Imagine you are a white hat hacker (the good kind) and have been commissioned to find flaws in a company's new cryptographic scheme. You've found one, but you don't want to reveal the details until after the company has paid you. You need a way to convince them that you're telling the truth given these restrictions. Describe a solution to this problem, and prove that it is complete, sound, and zero-knowledge.

Appendix B

Studio solutions

B.1 Studio 1.1 Solutions: Communication Systems

(1) What is Shannon's maxim?

Shannon's maxim tells us to err on the side of caution by assuming the the eavesdropper Eve knows how Alice and Bob have chosen to encrypt their messages. We also assume that Eve does not know the secret key that Alice and Bob have agreed upon.

(2) Convert the following text into its associated numeric vector:

Go, Bruins!

We can either do this by hand via individually converting each letter, or we can use the function `lettersToNumbers` provided.

```
>> lettersToNumbers('Go, Bruins!')

ans =

     6    14     1    17    20     8    13    18
```

Notice that `lettersToNumbers` takes out the comma, the space, and the exclamation point for us.

(3) Convert the text located in `two_cities_short.txt` into its associated vector of numeric values.

Since this text is so long, it's not really feasible to convert each character by hand. The `loadText` function converts a text file into its associated vector of numbers:

```
>> plaintext = loadText('../text/two_cities_short.txt');
```

Your file location could differ depending on where you've put your text files.

(4) Convert the following numeric characters into their associated letters:

```
[19,7,4,1,14,18,19,14,13,19,4,0,15,0,17,19,24]
```

We could convert each of these numbers by hand, or we could use the function `numbersToLetters` to help us out.

```
>> numbersToLetters([19,7,4,1,14,18,19,14,13,19,4,0,15,0,17,19,24])
THEBO STONT EAPAR TY
```

(5) Load the data found in the file `communication_systems_numbers_1.mat` by typing `load communication_systems_numbers_1.mat` in the command line of Matlab. You should now have a variable called `plaintext` in your workspace.

(You can verify that this variable is actually there by entering `whos` into the command line or by entering the variable name itself into the command line.) Convert the numeric values in `plaintext` into their associated letters.

We begin by loading the data. (Your file name could be different depending on where you've put your text files.)

```
>> load ../text/communication_systems_numbers_1.mat
```

We can then verify that the variable `plaintext` is in place.

```
>> whos
  Name            Size              Bytes  Class     Attributes

  plaintext       1x516              4128  double
```

In general, the Matlab function `whos` shows you all variables that are currently in your workspace.

We can then print the characters associated with the first 100 numbers located in `plaintext`

```
>> numbersToLetters(plaintext(1:100));
ALLST ATESA LLPOW ERSTH ATHAV EHELD ANDHO LDRUL EOVER MENHA
VEBEE NANDA REEIT HERRE PUBLI CSORP RINCI PALIT IESPR INCIP
```

It reads "All states all powers that have held and hold rule over men have been and are either republics or principalities princip..."

B.2 Studio 1.2 Solutions: Matlab

(1) Generate an array of whole numbers from 10 to 20. Multiply the array by 3.5, then add 7. What is the 11^{th} element of the final array?

```
EDU>> x = 10:20;
EDU>> y = 3.5*x + 7;
EDU>> y(11)

ans =

    77
```

(2) Write a function that takes an array `x` as an input and doubles every entry in `x` to form `y` as an output. Call your function `arrayDoubler`.

```
function y = arrayDoubler(x)
    y = 2*x;
end
```

(3) Write a function that takes an array `x` as an input and returns the entry in the middle of the array if the array has an odd number of entries and returns the entry just to the left of the middle of the array if the array has an even number of entries. (Hint: check out the `round` function)

```
function middleValue = getMiddleValue(x)
    middleIndex = round(length(x)/2);
    middleValue = x(middleIndex);
end
```

(4) The **factorial** of a number n is defined by $n! = n(n-1)(n-2)\cdots(2)(1)$. Write a function called `factorial` that accepts a number `n` as an input, and returns the quantity $n!$ as an output. (Hint: use a `for` loop!)

```
function y = factorial(n)
    y = 1;
    for i = 1:n
        y = y*i
    end
end
```

(5) Write a function that takes an input whole number `x`. If `x` is less than 10, your function should turn a matrix with 3 rows and 2 columns containing only

zeros. If `x` is greater than or equal to 10, your function should return a matrix with 2 rows and 3 columns containing only ones. (Hint: check out the functions `zeros` and `ones`.)

```
function mat = myFunc(x)

if x < 10
    mat = zeros(3,2);
elseif x >= 10
    mat = ones(2,3);
end
```

B.3 Studio 2.1 Solutions: Transposition Ciphers

(1) Encipher the plaintext "I made him some coffee" using a transposition cipher with key $k = 4$.

Using padding of all `A`s, the ciphertext reads

```
IHMFM IEEAM CEDSO AEOFA
```

(2) Encipher the plaintext found in `earnest_short.txt` using a transposition cipher with key $k = 64$.

Suppose we have a function called `transpositionEncrypter(plaintext, key)` which performs transpositional encryption on input `plaintext` using the input `key` as the key. The following commands then perform the desired encryption:

```
>> plaintext = loadText('../text/earnest_short.txt');
>> ciphertext = transpositionEncrypter(plaintext, 64);
```

The first 100 characters of ciphertext read

```
>> numbersToLetters(ciphertext(1:100))
FFSAO HLAAM ASCOE NTIER GSLNE TENRS DPRTL HADDS ARADA RWUKE
AOSTL OGHLY TNOEI TENWE GLRES OEBPO HSSYE HLBDH LEHEB LAOWT
```

(3) Decipher the ciphertext

```
IHSFI TOMAR HKGIE ISSVM AEIA
```

assuming a transposition cipher with key $k = 3$.

After removing the padding, the plaintext reads: "I FORGIVE HIM HIS MISTAKES."

(4) Decipher the ciphertext located in `transposition_ciphertext_1.txt` assuming a transposition cipher with key $k = 64$.

Suppose we have a function called `transpositionDecrypter(ciphertext, key)` which performs transpositional decryption on input `ciphertext` using the input `key` as the key. The following commands then perform the desired decryption:

```
>> ciphertext = loadText('../text/transposition_ciphertext_1.txt');
>> plaintext = transpositionDecrypter(ciphertext, 64);
```

The first 100 characters of ciphertext read

```
>> numbersToLetters(plaintext(1:100))
TOSHE RLOCK HOLME SSHEI SALWA YSTHE WOMAN IHAVE SELDO MHEAR
DHIMM ENTIO NHERU NDERA NYOTH ERNAM EINHI SEYES SHEEC LIPSE
```

(5) Decode the ciphertext located in `transposition_ciphertext_no_key_1.txt` assuming a transposition cipher assuming that you do not have knowledge of the key.

Let's assume that we have a piece of code called `transpositionDecrypter(ciphertext, key)` which deciphers the input `ciphertext` using the input `key`.

We're going to guess and check different keys in a systematic way. We know that the length of the ciphertext must be a multiple of the key. The length of the ciphertext is 572. The divisors of the ciphertext length are 1, 2, 4, 11, 13, 22, 26, 44, and so on. If the key were $k = 1$, the ciphertext would be the plaintext, which wouldn't be a very good thing. So the divisors 2 and larger are the only possible keys if the transposition enciphering has been performed correctly.

We could check each of these individually and arrive at the correct conclusion. We could also write a quick piece of code to automate this process. There is one possible solution:

```
function transpositionDecrypterTrialAndError(ciphertext)
for key = 2:20
   if mod(length(ciphertext),key) == 0
       key
       guess = transpositionDecrypter(ciphertext, key);
       numbersToLetters(guess(1:50)) % print only the first part of the plaintext
   end
end
```

The `if` statement at the beginning of the loop makes sure that the ciphertext length is a multiple of the key. If this condition does not hold, we continue on to the next iteration of the loop. Running this code on the cipher text produces the following:

```
key =  2
IRTOG TOEEO HATWT PEOYG ITSAL TSHMN PUTLE EDREE DTOIC HWTFA

key = 4
ITGOE HTTEY ISLSM PTEDE DOCWF SEEGH EHOSM FFTTL LNRGO HYHHE

key = 11
IIASS HLTCN HRSCI HOBTR TOHEH NYLAT IEFAD HHNHG EOHHI RAOUA

key = 13
ITWAS SEVEN OCLOC KOFAV ERYWA RMEVE NINGI NTHES EEONE EHILL
```

We can see that the first 3 attempts give gibberish as the plaintext. But $k = 13$ gives us a nice, English sentence as an output: "It was seven o'clock of a very warm evening in the Seeonee hill..." Notice that proper names can be a little confusing. The chances of this happening if $k = 13$ were in fact not the actually

key are very, very small. We can therefore conclude with a reasonable amount of certainty that this is indeed the plaintext.

B.4 Studio 2.2 Solutions: Caesar Ciphers

(1) Encrypt the following message using a Caesar cipher with key $k = 25$ by hand:

```
THEBA NKERO FFERS HERAB RIBE
```

Notice that adding 25 is the same thing as subtracting 1 modulo 26, because $-1 \equiv 25 \bmod 26$. Therefore, we shift each letter in the plaintext back by 1 in order to form the ciphertext.

```
SGDAZ MJDQN EEDQR GDQZA QHAD
```

(2) Encrypt the plaintext located in `text/jungle_short.txt` using a Caesar cipher with key $k = 4$ using code you have written.

The first 100 characters read

```
MXAEW WIZIR SGPSG OSJEZ IVCAE VQIZI RMRKM RXLIW IISRI ILMPP
WALIR JEXLI VASPJ ASOIY TJVSQ LMWHE CWVIW XWGVE XGLIH LMQWI
```

(3) Decrypt the following message assuming a Caesar cipher with key $k = 3$ by hand:

```
WKHBU HDGWK HFKLO GUHQD VWRUB
```

The plaintext reads

```
THEYR EADTH ECHIL DRENA STORY
```

(4) Decrypt the ciphertext located in `text/ciphertext_caesar_1.txt` assuming a Caesar cipher with key $k = 17$ using code you have written.

The first 100 characters of the plaintext read

```
THEPR INCEC HAPTE RIHOW MANYK INDSO FPRIN CIPAL ITIES THERE
AREAN DBYWH ATMEA NSTHE YAREA CQUIR EDALL STATE SALLP OWERS
```

(5) Decrypt the ciphertext located in `text/ciphertext_caesar_no_key_1.txt` assuming Caesar cipher but no knowledge of the key k.

Suppose we have a function `caesarDecrypter` which takes inputs `ciphertext` and `key` and produces an output `plaintext` as their names suggest they do. Since the key space is just $\{2, 3, \ldots, 25\}$, we need only use a `for` loop to cycle through the possible keys and monitor the output. When we see an output that reads in plain English, it is overwhelmingly likely that it represents the correct key being used.

```
function caesarDecrypterTrialAndError()

ciphertext = loadText('../text/ciphertext_caesar_no_key_1.txt');

for key = 2:25
    guess = caesarDecrypter(ciphertext, key);
    numbersToLetters(guess(1:50))
end
end
```

We see that when $k = 13$, the decryption function produces

```
INMAT HEMAT ICSAG ROUPI SASET OFELE MENTS TOGET HERWI THANO
```

(6) Show that $\{1, 2, \ldots, 6\}$ is a group under multiplication modulo 7.

We have identity element 1. We could confirm by testing all pairs that the product of any two elements taken modulo 7 is in the set. Associativity follows from the fact that multiplication of integers is associative. The hardest part is inverses. The set is small enough we can compute the inverses by trial and error:

$$1^{-1} = 1,\ 2^{-1} = 4,\ 3^{-1} = 5,\ 6^{-1} = 6.$$

Notice that if $3^{-1} = 5$, then $5^{-1} = 3$, so this list is complete. Also notice that we can get some perhaps unexpected behavior, such as an element being its own inverse. This is OK!

We conclude that the set $\{1, 2, \ldots, 6\}$ is a group under multiplication.

(7) Show that $\{1, 2, 3, 4, 5\}$ is not a group under multiplication modulo 6.

Taking $2 \cdot 3 \bmod 6$ we see the set is not closed under multiplication modulo 6. Moreover, we see that there is no element in the set that acts as an inverse for the element 2 (or the element 3 for that matter). We will see that these two facts are not unrelated. We conclude that the set is not a group under multiplication modulo 6.

(8) Suppose we have a group G. Show that a has a unique inverse b. (Hint: suppose a has two inverses, b and c such that $b \neq c$, and see if you can get a contradiction.)

Let's follow the hint and assume that $a + c \equiv 0$, $a + b \equiv 0$ and $b \neq c$. Then

$$\begin{aligned} b &\equiv b + 0 \text{ (definition of the identity)} \\ &\equiv b + (a + c) \text{ (a and c are inverses by assumption)} \\ &\equiv (b + a) + c \text{ (associativity)} \\ &\equiv c \text{ (a and b are inverses by assumption)} \end{aligned}$$

But we have assumed that $b \neq c$. Therefore, we have a contradiction that shows that our initial assumption that a has two distinct inverses must be incorrect.

(9) Is the set of all possible fractions (also called **rational numbers**), commonly denoted by the symbol $\mathbb{Q}$, a group under multiplication?

It would seem that everything checks out until we consider the element 0. Since the identity element under multiplication is 1, we would need for there to be a number x such that $x(0) = 1$ in order to full the invertibility requirement of groups. No such rational number exists.

B.5 Studio 2.3 Solutions: Affine Ciphers

(1) Is the key pair $(k, \ell) = (3, 2)$ appropriate in affine ciphering on the English alphabet?

If we stick with the notation that $(k, \ell) = (3, 2)$ produces ciphertext $\mathbf{c} = \ell\mathbf{p} + k$, then no, the pair is not appropriate. This arises from the fact that $gcd(\ell = 2, 26) = 2 > 1$. This implies that 2 is a zero divisor in $\mathbb{Z}_{26}$, and so has no inverse. Bob can therefore not uniquely decode the ciphertext into plaintext.

(2) Perform affine encryption assuming key $(k, \ell) = (5, 3)$ on the plaintext located in `text/rings.txt`.

The first 100 characters read

```
DSPFK ARPFK DLHFS OPVER HYRLD UDLFM MZDSF MXRIE FFEDS XDHFS
FMXRI EFDLH KENLK NERTD KAVYR EFKDV SHXRS REFMD CDSXK ARFED
```

(3) Perform affine decryption assuming key $(k, \ell) = (0, 7)$ on the ciphertext located in `text/ciphertext_affine_1.txt`.

The first 100 characters read

```
THEAF FINEC IPHER ISATY PEOFM ONOAL PHABE TICSU BSTIT UTION
CIPHE RWHER EINEA CHLET TERIN ANALP HABET ISMAP PEDTO ITSNU
```

The text above reads "The affine cipher is a type of mono alphabetic substitution cipher wherein each letter in an alphabet is mapped to its nu..."

(4) Perform affine decryption assuming no knowledge of the key on the ciphertext located in `text/ciphertext_affine_no_key_1.txt`

Suppose we have a function called `affineDecrypter` that does what you'd expect given our work so far. Then a brute force attack algorithm might look like the following:

```
function affineDecrypterTrialAndError()
clc
ciphertext = loadText('../text/ciphertext_affine_no_key_1.txt');

for k = 1:25
    for ell = 1:25
        if gcd(ell,26) == 1
            guess = affineDecrypter(ciphertext, k, ell);
            k, ell, numbersToLetters(guess(1:50))
        end
    end
end
end
```

This function produces a lot of output, and sifting through it is time-consuming. You might do a "Find" for common words like `THE` to see if any of the candidates make sense. We observe a promising candidate at $(k, \ell) = (7, 9)$.

```
k =

     7


ell =

     9

MEMPH ISISA CITYI NTHES OUTHW ESTER NCORN EROFT HEUSS TATEO
```

The text above reads, "Memphis is a city in the southwestern corner of the state o. . . ." Looks good!

(5) Show that $(ab) \bmod n \equiv (a \bmod n)(b \bmod n)$. (Hint: write $a = q_a n + r_a$ and similarly for b.)

We can write $a = q_a n + r_a$ and $b = q_b n + r_b$. Then

$$(ab) \equiv (q_a n + r_a)(q_b n + r_b) \tag{B.5.1}$$
$$\equiv q_a q_b n^2 + q_a n r_b + r_a q_b n + r_a r_b. \tag{B.5.2}$$

Since any multiple of n is equivalent to zero modulo n, the first three terms on the right are zero. The desired result follows directly:

$$(ab) \equiv r_a r_b \tag{B.5.3}$$
$$\equiv (a \bmod n)(b \bmod n). \tag{B.5.4}$$

(6) Show that the collection of all polynomial functions $\mathcal{P}$ in one variable x is a ring.

There are several things to verify here, but most of them are easy. First, note that the elements of this ring are polynomials functions, not numbers. That may look strange, since it's probably the first time you've seen it. But it's mathematically perfectly fine. In fact, thinking about functions in terms of rings can be very helpful, for instance in discussion of error control codes.

Suppose $p(x)$ and $q(x)$ are two polynomial functions. Their sum $p(x) + q(x)$ is a polynomial, as is their product $p(x)q(x)$. Therefore $\mathcal{P}$ is closed under both addition and multiplication. Moreover, both addition and multiplication are associative and commutative, as we know by the way we've always done polynomial arithmetic. There exists the multiplicative identity element 1, and the additive identity 0, both of which are polynomials, albeit very simple ones. There exist additive inverse of every polynomial $p(x)$, name $-p(x)$. Finally,

multiplication distributes across addition. (Note that there are not multiplicative inverses for every element, but we don't need this property for the set to be a ring.) We conclude that $\mathcal{P}$ is indeed a ring, and a commutative one at that.

(7) Form a complete table listing all the multiplicative inverse pairs in $\mathbb{Z}_{26}$. Clearly denote all zero divisors as well.

The zero divisors are 2, 4, 6, 8, 10, 12, 13, 14, 16, 18, 20, 22, and 24. The units and their inverses are $1^{-1} = 1$, $3^{-1} = 9$, $5^{-11} = 21$, $11^{-1} = 19$, $17^{-11} = 23$. Notice that since inverses come in pairs, we only need to list each number once. We could definitely compute these inverse pairs by hand, but another, perhaps easier way, would be with the provided function

```
>> modInv(17, 26)

ans =

    23
```

Varying `x` will produce the desired results.

(8) Show that every nonzero element in $\mathbb{Z}_p$ has a multiplicative inverse if p is prime.

By definition, a prime p is a number such that $\gcd(p, q) = 1$ for all $0 < q < p$. The property therefore follows directly from the fact (which we proved in the text) that if $\gcd(p, q) = 1$, then q has a multiplicative inverse in $\mathbb{Z}_p$.

(9) Show that while $\mathbb{Z}_n$ is never a multiplicative group, the collection of units $\mathbb{Z}_n^\times$ is always a multiplicative group.

The full ring is never a multiplicative group due to the presence of 0. The units are a group. We can verify the axioms:

- Multiplication amongst the units is associative because it is in the larger ring
- There exists an identity, namely 1
- By the definition of units, every element in the set has an inverse

The only axiom that remains to be verified is closure, namely the idea that a unit times a unit is another unit. Suppose we have two units a and b. We want to show that the product ab has an inverse $(ab)^{-1}$. Playing around for a while would produce $(ab)^{-1} = b^{-1}a^{-1}$. Let's verify

$$(b^{-1}a^{-1})(ab) \equiv b^{-1}(a^{-1}a)b \equiv b^{-1}b \equiv 1. \tag{B.5.5}$$

So the set of units is closed under multiplication. We conclude that $\mathbb{Z}_n^\times$ is a group under multiplication modulo n.

(10) We will show that an element is a zero divisor in $\mathbb{Z}_n$ if and only if it does not have an inverse in $\mathbb{Z}_n$.

(a) Suppose that an element x is a zero divisor in $\mathbb{Z}_n$. Show that x cannot have a multiplicative inverse. (Hint: follow the general strategy found taken in the notes.)

If x is a zero divisor, then there exists some nonzero y such that $xy \equiv 0 \bmod n$. Suppose for contradiction that x also has a multiplicative inverse x^{-1} such that $xx^{-1} \equiv 1 \bmod n$. Then

$$xy \equiv 0 \tag{B.5.6}$$

$$x^{-1}xy \equiv x^{-1} \cdot 0 \tag{B.5.7}$$

$$y \equiv 0. \tag{B.5.8}$$

But we had already assumed that y was nonzero. Therefore, it cannot be the case that x has a multiplicative inverse.

(b) Now suppose that x does not have a multiplicative inverse. Show that x must be a zero divisor. (Hint: think about xa_1, xa_2, ..., xa_n for all the elements $a_i \in \mathbb{Z}_n$, and remember that we've assumed that there is no a_i such that $xa_i \equiv 1$.)

Let's follow the hint's direction. We have $n-1$ different products of the form ae_i, where both a and e_i are nonzero. Since x doesn't have a multiplicative inverse, none of these products are equal to 1 modulo n. Therefore these products can assume only $n-2$ of the $n-1$ nonzero values in $\mathbb{Z}_n$. Since there are $n-1$ products and only $n-2$ possible values the products can assume, there must be at least two products that are the same value. Let's call these ae_i and ae_j, where $e_i \neq e_j$. Symbolically,

$$ae_i \equiv ae_j \tag{B.5.9}$$

$$a(e_i - e_j) \equiv 0. \tag{B.5.10}$$

The last line follows from the distributive property of rings. Since $e_i \neq e_j$, the difference $e_i - e_j$ is nonzero, and therefore a is a zero divisor.

B.6 Studio 2.4 Solutions: Polyalphabetic Ciphers

(1) Encrypt the plaintext "Cryptology" using Vigenère encryption with the key "KEY".

The ciphertext **c** is computed by adding vertically modulo 26 the numerical representations of the array

```
CRYPTOLOGY
KEYKEYKEYK
```

We conclude that the ciphertext reads.

```
MVWZX MVSEI
```

(2) Decrypt the ciphertext

```
DLCFM EORCB IASTF OV
```

assuming Vigenère encryption with key "KEY".

The plaintext **p** is computed by subtracting the second line of the following array from the first modulo 26:

```
DLCFMEORCBIASTFOV
KEYKEYKEYKEKEYKEY
```

We conclude the plaintext reads "The Vigenere cipher."

(3) Encrypt the plaintext located in the file `text/cryptography_short.txt` using the Vigenère cipher with key "UNBREAKABLE".

The first 50 characters read

```
WEZGX OQRBA LSBST VYZTP WSALG ISMQR FPOJG IZHDO NTPGL RURRD
```

(4) Decrypt the ciphertext located in the file `text/ciphertext_polyalphabetic_1.txt` assuming Vigenère encryption with key "UNBREAKABLE".

The first 50 characters read

```
ITWAS THESE ASONO FDARK NESSI TWAST HESPR INGOF HOPEI TWAST
```

We conclude the plaintext begins "It was the reasons of darkness it was the spring of hope it was t..."

(5) Encrypt the plaintext located in `text/memphis.txt` using a one-time pad with the text in `text/yankee.txt` as the key.

The first 50 characters of the ciphertext read

```
UEYPU IEMJI EIGGE NLISJ BUGKN ISKIU VPVRE XWCWW PRNZW LTTXS
```

(6) Decrypt the ciphertext located in `text/ciphertext_polyalphabetic_2.txt` assuming encryption with a one-time pad with the text in `text/yankee.txt` as the key.

The first 50 characters of the plaintext read

```
INMAT HEMAT ICSAN DMORE SPECI FICAL LYINA LGEBR AARIN GISAN
```

(7) Decrypt the ciphertext located in `text/ciphertext_polyalphabetic_no_key_1.txt` assuming Vigenère encryption with no knowledge of the key.

B.7 Studio 3.1 Solutions: Diffie-Hellman

(1) Is 2 primitive modulo 13?

If all numbers less than 13 are coprime with 13, it must be the case that every number from 1 to 12 can be expressed as a power of 2 modulo 13. We can simply compute all the powers and see if this is true. We can perform each computation individually, or use Matlab to compute them all at once.

```
>> mod(2.^[1:12],13)

ans =

     2     4     8     3     6    12    11     9     5    10     7     1
```

Recall that `.^` performs element-wise exponentiation in Matlab. Also note that this code will not execute without the seemingly innocuous period before the circumflex.

(2) Is 3 primitive modulo 13?

If all numbers less than 13 are coprime with 13, it must be the case that every number from 1 to 12 can be expressed as a power of 3 modulo 13. We can simply compute all the powers and see if this is true. We can perform each computation individually, or use Matlab to compute them all at once.

```
>> mod(3.^[1:12],13)

ans =

     3     9     1     3     9     1     3     9     1     3     9     1
```

Recall that `.^` performs element-wise exponentiation in Matlab. Also note that this code will not execute without the seemingly innocuous period before the circumflex.

Notice how the powers of 3 begin to repeat modulo 13. In fact, after we see the element 1, we know the sequence will repeat. Therefore, we can be sure that some elements, like 2 for instance, cannot be expressed as a power of 3 modulo 13. We conclude that 3 is not a primitive root in $\mathbb{Z}_{13}$.

(3) Assume that Alice and Bob perform Diffie-Hellman key exchange using prime $p = 61$ and primitive element $g = 10$. Imagine that Alice's private key is $a = 7$ and Bob's is $b = 50$. Compute the public message Alice sends Bob, the public message Bob sends Alice, and the shared secret number s at the end of the exchange.

By the definition of the Diffie-Hellman protocol, we have $A \equiv g^a \bmod p$, $B \equiv g^b \bmod p$, and $s \equiv g^{ab} \equiv B^a \equiv A^b \bmod p$. The numbers here are getting pretty big (*e.g.* 26^{50}), so we should use `powMod` rather than `mod` in order to avoid the potential for problems.

```
>> A = powMod(10,7,61)

A =

    26

>> B = powMod(10,50,61)

B =

    48

>> s = powMod(26,50,61)

s =

    48

>> s = powMod(48,7,61)

s =

    48
```

(4) Assume that Alice and Bob perform Diffie-Hellman key exchange using prime $p = 997$ and primitive element $g = 11$. Imagine Alice's private key is $a = 812$ and Bob's is $b = 903$. Compute the public message Alice sends Bob, the public message Bob sends Alice, and the shared secret number at the end of the exchange.

Recall that $A \equiv g^a \bmod p$, $B \equiv g^b \bmod p$, and the shared secret is $s \equiv g^{ab} \equiv B^a \equiv A^b$. The numbers here are quite large, so we'll have to use `powMod`.

```
>> A = powMod(11,812,997)

A =

   321

>> B = powMod(11,903,997)

B =

   979

>> s = powMod(321,903,997)
```

```
s =

   185

>> s = powMod(979,812,997)

s =

   185
```

(5) Find two widely accepted technologies, current or formerly in use, that use Diffie-Hellman key exchange.

Transport Layer Security (TLS), one of the foundational cryptographic protocols that underlies the Internet, uses Diffie-Hellman to generate keys shared between client and server.

Secure Shell (SSH), a program used to remotely log into computers, uses Diffie-Hellman.

Diffie-Hellman is also used in Internet Protocol Security (IPSec), a collection of technologies that underlie many internet layer communications.

B.8 Studio 3.2 Solutions: RSA

(1) Show that every zero divisor has infinite order. (Hint: suppose it has order k and try to find a contradiction.)

If a zero divisor x had finite order k, then by definition $x^k \equiv 1$. This would imply that $x(x^{k-1}) \equiv 1$, meaning that $x^{k-1} \equiv x^{-1}$. This is impossible, since x is a zero divisor.

(2) Show that every unit in $\mathbb{Z}_n$ has order less than or equal to $\phi(n)$.

Suppose a unit x does not have order less than or equal to $\phi(n)$. Then none of the elements

$$x^1,\ x^2, \ldots,\ x^{\phi(n)}$$

are equivalent to 1 modulo n. Each x^i is a unit, since x is a unit; we could convince ourselves that $(x^i)^{-1} \equiv (x^{-1})^i$. But none of the x^i are the unit 1. Therefore, since we have $\phi(n)$ powers of x and only $\phi(n)-1$ values these powers could take on, we know that two of them are equivalent, that is $x^i \equiv x^j$ for some $1 \leq i < j \leq \phi(n)$. Then

$$x^i \equiv x^j \tag{B.8.1}$$

$$x^i - x^j \equiv 0 \tag{B.8.2}$$

$$x^i(1 - x^{j-i}) \equiv 0 \tag{B.8.3}$$

The exponent $j - i$ is less than $\phi(n)$, so by assumption that x does not have order less than or equal to $\phi(n)$, the second factor in the product is nonzero; the only way $x^{j-i} \equiv 1$ would be for $j - i$ to be some multiple of $\phi(n)$. But this would imply that x is a zero divisor, a contradiction, since we assumed from the outset that x is a unit.

(3) Show that $\phi(p^2) = p^2 - p$ if p is prime.

There are $p^2 - \phi(n)$ zero divisors in $\mathbb{Z}_{p^2}$. We'll count the zero divisors and then do the conversion to the number of units. The only divisors of p^2 are p and 1. Therefore, the only zero divisors in $\mathbb{Z}_{p^2}$ are all the multiples of p, namely

$$0,\ p,\ 2p,\ 3p,\ \ldots, (p-1)p.$$

So there are exactly p zero divisors. This gives us $p^2 - p$ units. We conclude $\phi(p) = p^2 - p$.

(4) Prove the following claim: if an element x has order $\phi(n)$ in $\mathbb{Z}_n$, then x is a primitive element in $\mathbb{Z}_n$.

Recall that an element x is primitive in $\mathbb{Z}_n$ if every unit in $\mathbb{Z}_n$ can be written as a power of x, that is, there exists i so that $y = x^i \bmod n$ for every unit y in $\mathbb{Z}_n$.

Since $x^{\phi(n)} \equiv 1 \bmod n$, we have $xx^{\phi(n)-1} \equiv 1 \bmod n$. This implies that $x^{-1} \equiv x^{\phi(n)-1}$. This implies that x is a unit in $\mathbb{Z}_n$. Similar reasoning shows x^k

is a unit in $\mathbb{Z}_n$, because $1 \equiv x^{\phi(n)} \equiv x^k x^{\phi(n)-k}$. So every power of x is a unit. It remains to show that by taking all powers of x, we can recover *all* the units of $\mathbb{Z}_n$. In other words, we need to show that the elements

$$x^1,\ x^2,\ \ldots,\ x^{\phi(n)}$$

must all be different. To see this, suppose that $x^i \equiv x^j \bmod n$ for some pair $i < j \leq \phi(n)$. Then

$$x^i \equiv x^j \tag{B.8.4}$$

$$x^i - x^j \equiv 0 \tag{B.8.5}$$

$$x^i(1 - x^{j-i}) \equiv 0 \tag{B.8.6}$$

Since x^i is a unit, it is not a zero divisor. Therefore, for this equation to hold, it must be the case that $1 - x^{j-i} \equiv 0 \bmod n$. But this would imply that x has order less than or equal to $j - i$ which is less than $\phi(n)$ by our choice of i and j. This contradicts our assumption that the order of x is $\phi(n)$. We conclude that all the powers $x^1, x^2,\ \ldots,\ x^{\phi(n)}$ must all be distinct. Since there are exactly $\phi(n)$ units in $\mathbb{Z}_n$, every unit must be expressible as a power of x. Therefore x is primitive in $\mathbb{Z}_n$.

(5) Imagine an RSA cryptosystem is created with primes $p = 991$ and $q = 113$.

(a) Imagine Alice chooses $e = 50$. Is this value appropriate? Why or why not?

We need e to be a unit in $\mathbb{Z}_{\phi(n)}$. We can verify whether $e = 50$ is a unit in $\mathbb{Z}_{\phi(991\cdot 113)}$ with Matlab.

```
EDU>> gcd(50,990*112)
ans =

    10
```

(Recall that $\phi(pq) = (p-1)(q-1)$.) Since the greatest common divisor is greater than 1, the element $e = 50$ is not a unit in the appropriate ring. Alice should not use $e = 50$.

(b) Imagine Alice now chooses $e = 53$. Verify that this choice is acceptable. What information does Alice publish?

Here, $e = 53$ is a unit in $\mathbb{Z}_{\phi(991\cdot 133)}$.

```
EDU>> gcd(53,990*112)
ans =

     1
```

Alice publishes both $e = 53$ and $n = pq = 111983$.

(c) Imagine Bob wants to send the message $m = 100$. Is this value OK? Why or why not?

This is perfectly fine. Bob's message should be an element of $\mathbb{Z}_n$. It does not need to be a unit, so for instance $m = 1982$ would be a perfectly fine choice, too.

(d) What ciphertext does Bob send across the channel given the numbers we've decided on so far?

In general, Bob sends $c \equiv m^e \bmod n$. Here, we have

```
EDU>> c = powMod(100,53,111983)
c =

       49329
```

(6) Imagine Bob has chosen $p = 937$, $q = 683$, and $e = 383179$.

(a) Is Bob's choice of e appropriate? Why or why not?

Bob's choice of e must be a unit in $\mathbb{Z}_{\phi(n)}$. Since p and q are both primes, we have $\phi(pq) = (p-1)(q-1)$. We can test whether e is a unit in $\mathbb{Z}_{\phi(n)}$:

```
>> gcd(383179, 936*682)
ans =

     1
```

We conclude that e is a unit in $\mathbb{Z}_{\phi(n)}$, and therefore Bob's choice of e is appropriate for RSA.

(b) Alice sends Bob $c = 170133$. What is the associated plaintext?

In general, we can assume that Alice has sent Bob $c \equiv m^e \bmod n$. To recover the plaintext m, Bob must first compute the multiplicative inverse of e in $\mathbb{Z}_{\phi(n)}$:

```
>> modInv(383179, 936*682)
ans =

     447379
```

Bob then computes $c^{e^{-1}} \equiv m^{ee^{-1}} \equiv m^{1+k\phi(n)} \equiv m \bmod n$, where the last equivalence follows from Euler's theorem. Numerically, this results in

```
>> powMod(170133, 447379, 937*683)
ans =

     101
```

B.9 Studio 3.3 Solutions: Cryptographic Hashes

(1) Generating truly random bits is very difficult. In practice, we often settle for *pseudorandom bits*, meaning that the bits "look" random but are generated by a deterministic (*i.e.*, non-random) procedure. Explain how a hash function could be used to generate pseudorandom numbers?

Since a hash function takes an input and produces a "random" output, we might think that this is a good place to start. Even if an attacker knew the general type of inputs we were using, predicting the exact output should be expected to be hard. More quantitatively, if we wanted to generate k random bits, we should choose a hash function H which produces an output of $n \geq k$ bits. We can then convert the any k bits we choose into a number between 0 and $2^k - 1$.

(2) Passwords are often hashed before they are stored on a company's website.

(a) Imagine you provide your password when you try to login to a website. The web server will compare the password you provide to the hashed password it has associated with your account. Use the properties of hashes to explain why this procedure will allow you to login if and only if you've provided the correct password (with very high probability).

Collision resistance tells us that it is almost impossible that you could pick a false password that has the same hash as your original password; the website will only let you login when you provide the correct password. If you provide the correct password, it will produce the correct hash, because there is no randomness in a hash function, *i.e.*, a given input always leads to the same output.

(b) Why wouldn't the company just compare the password you provide to a plaintext version of your password it stores on its servers?

If a hacker were ever to gain access to the server on which the passwords were stored, the hacker would have direct access to all passwords. This is bad. But since people often reuse the same username/password combinations on different sites, a failure like this is particularly bad, because it makes other sites much more vulnerable.

(c) Experienced hackers know that companies store their passwords in hashed form. To speed up their attacks, hackers will compute the hashes of a bunch of common passwords (*e.g.*, "password", "1234", "asdf"). This collection of pre-computed hashes is called a *rainbow table*. (Don't believe the silly name? Check out https://www.freerainbowtables.com). Suppose a hacker manages to obtain a company's list of hashed passwords. How can a rainbow table be used to further compromise security?

Say the first entry in the rainbow table is the password `password` and its associated hash h. The hacker can scan the list of all hashed passwords in the company's list of hashed passwords. If the hacker finds a match, collision

resistance tells us that the match is most certainly generated because the user also chose password `password`. If the two hashes match but the passwords are not the same, then the user has found a collision in the hash.

Note that this is a particularly strong attack, because the hacker can look at the entire database at once and find *all* users that have a particular password.

(d) To mitigate the effectiveness of rainbow attacks, many websites use *salts*. A salt is a random number that is appended before your password before hashing. For a concrete example, imagine that you register at a new website. The server will generate a random number and assign this number as your account's salt. The server will then store $H([\text{salt password}])$. Explain how the use of salts makes rainbow tables far less effective, even if the hacker happens to know the salt that was applied to each password.

Again, let's assume the first entry in the rainbow table is the password `password` and its associated hash h. Imagine that the i^{th} user has salt s_i. To see whether the i^{th} user has password `password`, the hacker will have to compute $H([s_i$ `password`$])$ for every user. Since different users have different salts, even if two users have the same password, they will have different salted hashes. This forces the hacker to crack each user's password individually, rather than comparing all at once a single hashed password in the rainbow table to all hashed passwords in the company's list.

(3) One of the problems with using digital photography for collecting criminal evidence is the possibility of tampering and digital editing. Explain how a cryptography hash could be used to show a photo has not been altered since it was taken. (Hint: feel free to use the time stamp of when the photo was taken.)

The camera captures a time stamp when the image is taken (and perhaps even information about who was using the camera if the device is sufficiently sophisticated). The camera itself then computes the hash $H(\text{image stamp}])$. We could easily verify after the fact that the image was taken at the given time by recomputing the hash and making sure that the new hash matched the old hash. Moreover, it would be nearly impossible for someone to forge a picture that would have the same hash due to collision resistance.

(4) In a silent auction, bidders write down their bids for an item and submit these bids to the auctioneer. At the end of the silent auction, the highest bid wins. Hence, there is a strong incentive to be the highest bid, but as little as possible over the runner-up.

One problem with silent auctions is that the auctioneer can collude with an accomplice by telling her the highest bid. This allows the accomplice to win the auction without overspending by any more than they must.

It would be great if each bidder had to *commit* publicly to their bid. The problem is that if they reveal their bid directly, it defeats the point of a silent auction. Devise a scheme using cryptographic hashes that allows a bidder to show the public that they have locked in their bid. Your scheme should also

prevent other bidders from figuring what exactly the bid was, even if they know, for instance, that the only possible bids are increments of $10.

When the time comes to commit, each bidder chooses a random number, call it s. They then compute $H([s \text{ bid})$ and release this hash value to the public. When the time comes to show their bid, the bidder recomputes the hash and shows the public that the new hash and the old hash match. This implies that they have probably not changed their bid; after all, if they had found two [$s$bid] pairs that led to the same hash, they will have found a collision.

The random salt s makes it more difficult for other bidders to guess the bid the corresponds to a particular hash. Since a hash is a one-way function, no one could directly "undo" the hash to arrive at the original bid. That being said, if all bidders knew that bids were in increments of 10 dollars, it would be too hard to simply try every possible bid until you found a hash that matched. As in the password problem earlier, the salt doesn't completely fix this problem, but it makes cracking someone's bid much, much harder.

B.10 Studio 3.4 Solutions: Digital Signatures

(1) Imagine Alice wants to send a signed version of the message `Meet at midnight` to Bob using the RSA-based signature scheme.

(a) Use `sha0` to compute the message digest h of Alice's message.

```
>> h = sha0('Meet at midnight')

h =

        2646
```

(b) If Alice chooses $p_a = 911$, $q_a = 937$ and $e_a = 11$, what signature s associated to the given message does she send Bob?

Alice sends $s \equiv h^{e_a^{-1}} \bmod n_a$. Recall that we mean $e_a^{-1} e_a \equiv 1 \bmod \phi(n_a)$. In Matlab,

```
>> eaInv = modInv(11, 910*936)

eaInv =

      309731
```

Therefore, Alice sends signature

```
>> s = powMod(2646, 309731, 911*937)

s =

      404684
```

(2) Imagine now that Alice wants to send the numeric message $m = 123456789$ to Bob using an RSA-based signature scheme. Imagine that Alice has chosen $p_a = 911$, $q_a = 937$ and $e_a = 11$, and that Bob has published $n_b = 27491$ and $e_b = 13$.

(a) Use `sha0` to compute the message digest h of Alice's message.

```
>> sha0(123456789)

ans =

       32666
```

(b) What is the ciphertext that Alice sends Bob?

Alice sends $c \equiv m^{e_b} \bmod n_b$. Here, we have

```
>> c = powMod(123456789, 13, 27491)

c =

       24308
```

(c) What is the signature that Alice sends Bob?

In general, Alice's signature is $s \equiv h^{e_a^{-1}} \bmod n_a$. Remember that here e_a^{-1} means the inverse of e_a in $\mathbb{Z}_{\phi(n_a)}$, not in $\mathbb{Z}_{n_a}$. We have

```
>> eaInv = modInv(11, (911-1)*(937-1))

eaInv =

      309731
```

Then the signature Alice sends Bob is

```
>> s = powMod(2646, 309731, 911*937)

s =

      404684
```

(3) Imagine Alice has published $n_a = 853607$ and $e_a = 11$, and Bob has chosen $p_b = 37$, $q_b = 743$, and $e_b = 13$, both using the RSA scheme. Bob receives $c = 2409$ and $s = 790832$ from someone claiming to be Alice. Should Bob believe that this message actually came from Alice?

Bob begins by deciphering the ciphertext using $m \equiv c^{e_b^{-1}} \bmod n_b$. Here, the quantity e_b^{-1} is the inverse of e_b in $\mathbb{Z}_{\phi(n_b)}$, not in $\mathbb{Z}_{n_b}$. In Matlab,

```
>> ebInv = modInv(13, 36*742)

ebInv =

       18493
```

Then the message is given by

```
>> m = powMod(2409, 18493, 37*743)

m =

        6283
```

The hash of this message is

```
>> sha0(6283)

ans =

       12473
```

Bob needs to check this digest against the digest encoded Alice's signature. He can recover this hash digest via $h \equiv s^{e_a} \bmod n_a$. In Matlab,

```
>> h = powMod(790832, 11, 853607)

h =

       12473
```

Since the two values match, Bob can be very sure that Alice actually sent the message he received. Notice that in this problem, Bob had no knowledge of Alice's private information.

(4) Imagine Alice has published $n_a = 853607$ and $e_a = 11$, and Bob has chosen $p_b = 37$, $q_b = 743$, and $e_b = 13$, both using the RSA scheme. Bob receives $c = 182$ and $s = 46937$ from someone claiming to be Alice. Should Bob believe that this message actually came from Alice?

Bob begins by deciphering the ciphertext using $m \equiv c^{e_b^{-1}} \bmod n_b$. Here, the quantity e_b^{-1} is the inverse of e_b in $\mathbb{Z}_{\phi(n_b)}$, not in $\mathbb{Z}_{n_b}$. In Matlab,

```
>> ebInv = modInv(13, 36*742)

ebInv =

       18493
```

Then the message is given by

```
>> m = powMod(182, 18493, 37*743)

m =

        3141
```

The hash of this message is

```
>> sha0(3141)

ans =

       3378
```

Bob needs to check this digest against the digest encoded Alice's signature. He can recover this hash digest via $h \equiv s^{e_a} \bmod n_a$. In Matlab,

```
>> h = powMod(46937, 11, 853607)

h =

      748480
```

Since the two values do not match, Bob can be very sure that Alice did *not* actually send the message he received.

(5) Imagine Alice wants to send Bob a signed version of the message `Meet at midnight` using the DLP-based signature scheme with primitive element $g = 7$ in $\mathbb{Z}_{937}$. Assume that Alice and Bob have agreed to use the `sha0` hash. Alice's private key is $a = 117$, and for this transmission, she has chosen $k = 17$. What information does Alice send Bob?

In the DLP-based scheme, Alice sends $r \equiv g^k \bmod p$ and $s \equiv (h - ar)k^{-1} \bmod (p-1)$. We can compute r in Matlab using the ideas we've assembled:

```
>> r = powMod(2, 17, 937)

r =

   829
```

To compute s, we first need to compute the message digest h:

```
>> h = sha0('Meet at midnight')

h =

        2646
```

Here, the quantity k^{-1} is computed in $\mathbb{Z}_{p-1}$, not in $\mathbb{Z}_p$. We have

```
>> kInv = modInv(17, 936)

kInv =

   881
```

Combining the last two pieces, we can compute s:

```
>> s = mod((h - 117*r)*kInv,p-1)

s =

   837
```

(6) Imagine now that Alice wants to send Bob a signed version of the message $m = 1234$ using the ElGamal signature scheme. Assume that Alice and Bob have agreed on $g = 2$, $p = 1117$, and `sha0` as their hashing algorithm.

(a) Show that Alice and Bob's choice of g is appropriate given their choice of p.

To make Eve's cryptanalysis as difficult as possible, we need for g to be a primitive element in $\mathbb{Z}_p$. We can verify that the order of g is $p - 1$:

```
>> ord(2, 1117)

ans =

        1116
```

(b) Imagine that Alice has published $A = 501$ as her public key. If Bob receives message $m = 1234$ and signature $(r, s) = (539, 37)$ from a person claiming to be Alice, can Bob conclude that the message actually came from Alice?

Bob needs to confirm that $g^{H(m)}$ is equal to $r^s A^r$ modulo p. The hash digest is

```
>> h = sha0(1234)

h =

       62497
```

Then g^h is

```
>> powMod(2, 62497, 1117)

ans =

     2
```

We can also compute the second quantity:

```
>> mod(powMod(539, 37, 1117)*powMod(501, 539, 1117), 1117)

ans =

     2
```

Since the two quantities are equal, Bob can be very sure that the person who sent the message is actually Alice.

(c) Imagine that Alice has published $A = 501$ as her public key. If Bob receives message $m = 5678$ and signature $(r, s) = (539, 542)$ from a person

claiming to be Alice, can Bob conclude that the message actually came from Alice?

Bob needs to confirm that $g^{H(m)}$ is equal to $r^s A^r$ modulo p. The first quantity is given by

```
>> powMod(g, sha0(5678), 1117)

ans =

   531
```

The second quantity is given by

```
>> mod(powMod(539, 542, 1117)*powMod(501, 539, 1117), 1117)

ans =

   832
```

Since the two quantities are different, Bob can be very certain that the person who is sending the message is not actually Alice.

(d) Imagine that Alice has published $A = 501$ as her public key. Suppose Eve has determined that Alice is failing to choose a new value of k each time she sends a message to Bob. If Eve has collected two hash-signature pairs $(h_1, s_1) = (12473, 69)$ and $(h_2, s_2) = (3378, 446)$, what is Alice's private key a?

We know $s_1 k + ar = h_1 \bmod (p-1)$ and $s_2 k + ar = h_2 \bmod (p-1)$. Subtracting the second equation from the first gives

$$(s_1 - s_2)k \equiv h_1 - h_2 \bmod (p-1) \tag{B.10.1}$$
$$(69 - 446)k \equiv (12473 - 3378) \bmod 1116 \tag{B.10.2}$$
$$739k \equiv 9095 \bmod 1116 \tag{B.10.3}$$
$$k \equiv 739^{-1} 9095 \bmod 1116 \tag{B.10.4}$$
$$\equiv 595(9095) \bmod 1116 \tag{B.10.5}$$
$$\equiv 41. \tag{B.10.6}$$

This implies that $r \equiv g^k \bmod p \equiv 2^{41} \bmod 1117$. We can easily recover r in Matlab:

```
>> powMod(2, 41, 1117)

ans =

   539
```

We know now every quantity in the equation $s_1 k + ar = h_1 \bmod (p-1)$ except the private key a. We can rearrange the equation to solve for a:

$$a = r^{-1}(h_1 - s_1 k) \bmod (p-1).$$

Substituting in Matlab reveals the private key:

```
>> mod(modInv(539, 1116)*(12473 - 69*41), 1116)

ans =

   256
```

We can actually confirm that this is Alice's private key by checking it against her public key $A = 215$. We see that indeed $g^a = 2^{256} \bmod p = 501$.

Notice that this procedure could've failed in several places. First, if $(s_1 - s_2)$ had failed to be invertible in $\mathbb{Z}_{p-1}$, we would not have been able to solve for k. Second, if r had failed to be invertible in $\mathbb{Z}_{p-1}$, we would not have been able to solve for a. In conclusion, not choosing a new k after every message will not necessarily allow Eve to determine the private key. That being said, this sloppiness could seriously weaken the overall strength of the system.

B.11 Studio 3.5 Solutions: Zero-Knowledge Proofs

(1) A friend of yours is colorblind and so cannot tell his red socks from his green socks. In fact, he believes that they are actually the same color. You keep trying to tell him that the colors are different, that he looks like a fool with them on, and that he needs to get rid of them. Your friend thinks that you're trying to play a joke on him with all of this "different colors" business.

Fortunately, you've devised a plan to break the stalemate. You give him both a red and a green sock, and clearly tell him which is which. You tell him to randomly choose whether or not to switch the socks behind his back. He then reveals each hand to you, and you tell him which hand contains which sock. You can repeat the process over and over until your friend is sure you're telling the truth.

(a) Argue that this scheme is complete.

Recall that completeness means that if the prover is telling the truth, then the prover can convince the verifier. If you can actually distinguish the red sock from the green sock, you'll have no problem correctly answering your friend each round.

(b) Argue that this scheme is sound.

Recall that soundness means that if the prover is not telling the truth, then the verifier will be able to find out with high probability. If you cannot actually distinguish the red sock from the green sock, then the best you can do each round is guess. Following this strategy, you have a probability $1/2$ chance each round of answering correctly. If your friend each round independently randomizes which hand contains the red sock and which contains the green, then the probability of your answering correctly over k rounds is $(1/2)^k$. By varying the number of rounds, your friend can achieve any level of certainty up to but not including 100%.

(c) Argue that this scheme is zero-knowledge.

Assuming that the socks are exactly identical aside from their color, you cannot possibly convey any information about how you are distinguishing the two to your colorblind friend.

(2) Imagine you are a white hat hacker (the good kind) and have been commissioned to find flaws in a company's new cryptographic scheme. You've found one, but you don't want to reveal the details until after the company has paid you. You need a way to convince them that you're telling the truth given these restrictions. Describe a solution to this problem, and prove that it is complete, sound, and zero-knowledge.

Suppose we've found a way to break their scheme. We can ask the company to encrypt any message and send it over the channel. They keep the message hidden at this point. We break their encryption and recover the message. We reveal the message to them, and they confirm that our message matches theirs.

This scheme is complete, because if we have a way to break the encryption they're proposing, we can exactly follow the scheme above to convince them of this fact.

The scheme is sound, because if we do not have a way to break their encryption, there is virtually no chance we will be able to guess the message they sent. Even in the simplest case, there are two possible messages, and so we would only have a 1/2 probability of guessing correctly. By increasing the number of rounds as we've seen in other examples, the company can be as sure as they would like to be that we are telling the truth.

The scheme is zero-knowledge, because by simply revealing the product of our cryptanalytic process, namely the recovered message, we are not revealing any information about the process itself.

Appendix B. Studio solutions

Appendix C

Teaching Notes

C.1 Teaching Note 1.1: Communication Systems

While we use various digital communication schemes every day, these systems have become so transparent to the end user that few give a thought to how data is so reliably passed back and forth. This section is meant to give an introduction to the formalization of communication systems developed in the mid-1900s by Shannon and others. In particular, the three main phases of digital communication – compression, encryption and protection – each are briefly outlined through a non-technical example. While the course notes that follow will deal primarily with secrecy technologies, a broad and shallow understanding of the larger system should help students put what they learn in context.

The lesson also introduces the three major players in both classical and modern cryptosystems: the encrypter Alice, the decrypter Bob, and the eavesdropper Eve. These abstractions are incredibly convenient, both in the sense that they clearly delineate roles, but also in that they easily frame exercises for students. For instance, students can "be Alice" when encrypting and "be Eve" when breaking a cipher. This personification should help students make the critical distinction in each role between information they have and information they don't.

The lesson concludes with a discussion of numeric representations of text. The studio problems focus on building a familiarity with the functions provided with these notes for converting between numeric and character representations. This is a key skill, as the format in which the plaintext or ciphertext is given may not be the same as the format which the encrypter or decrypter expects.

Learning Objectives

After the lesson, students should be able to

- Sketch a rough diagram of the full communication system
- Articulate the roles of Alice, Bob, and Eve in the communication framework
- Articulate the importance of compression, encryption, and protection in modern communications
- Convert between numeric and character representations of text

C.2 Teaching Note 1.2: Matlab

This lesson is aimed at providing a very basic introduction to foundational programming concepts and Matlab syntax. The connections between Matlab and traditional calculators are first developed. Next, matrices and vector implementations in Matlab are defined and explored, including basic arithmetic, indexing, and slicing. (Note that matrix multiplication is not discussed in this section, except that `v*v`, with `v` a vector will not produce the element-wise multiplication that students might expect.) As the examples become more complex, scripts and functions are introduced in order to preserve work completed. It has proved helpful to have students follow along with simple examples, pausing every so often to help students debug as syntax errors are common. Next, `for` loops are covered in the context of automating repetitive tasks. Finally, `if`, `elseif`, and `else` are introduced for flow control. Here, one approach is to present the entire `if` block located on page 10 and allow students to figure out for themselves how things work. A gate question here is "What happens when `i = 25`?" The lesson concludes with a brief presentation of the specialized functions provided along with these course notes; this subsection could be left out and dealt with in a just-in-time fashion as the functions become necessary in other lessons.

The studio problems associated with this lesson begin with simple array manipulations. Students are then asked to create simple functions to remind themselves of the syntax framework associated with function definitions in Matlab. Ideally, they could use the example they completed in the notes as a reference. Students must learn about basic Matlab functions, including `round`, `ones`, and `zeros`. In addition, these problems are designed to encourage students to use the `help` function early and often as they learn the language. In a more challenging problem on loops and assignment, students are asked to write a function which computes the factorial of an input `n`. The studio concludes with an exercise on flow control via `if` statements.

There are many additional areas for exploration and assessment related to these ideas. For instance, implementing a function which computes the k^{th} partial sum of a geometric series with ratio a is a challenging extension of the factorial studio problem. There are numerous possibilities for assessment of flow control, the most challenging of which would be conditionals which are not mutually exclusive but which have some order of precedence.

Learning Objectives

After the lesson, students should be able to

- Explain the concept of variable assignment
- Structure the syntactical framework of a function without using references
- Write `for` loops without using references
- Use `if`, `elseif`, and `else` to control flow

C.3 Teaching Note 2.1: Transposition Ciphers

This lesson covers the classical transposition cipher. This family of ciphers has a rich historical context, including the Greeks' use of scytales on the battlefield, the deployment of the rail cipher in the American Civil War, and its combination with fractionation in more modern ciphers. The instructor can use scytale encryption as a hands-on ice-breaker activity.

In addition to the cryptographic content, this lesson is meant provide a gentle introduction to Matlab implementation of cryptographic techniques; matrices and vectors are introduced and connected to representation of plaintexts and ciphertexts, and the specialized functions provided with these course notes are employed. Basic Matlab operations like `reshape` and `transpose` are introduced.

The studio problems associated with this lesson (Section A.3 on page 76) require that students implement the transposition cryptosystem in code. (It is also expected that students will be able to perform encryption/decryption of simple messages by hand.) The `for` loop implementation of the brute force attack may be challenging for students. Section 1.2 provides a more complete discussion of `for` loops and their syntax, including a nested `if` statement that controls flow through the loop. The introduction of `mod` is also intentional, as it will play an integral role in much of the remainder of the course.

While this lesson only covers the classical transposition cipher, rail ciphers and more complicated variants are excellent opportunities for open-ended assessment, and fractionation provides a natural connection between classical cryptosystems and digital representation. Writing code for more complicated transpositional methods can quickly become intractable for beginner programmers. One variant that would only involve good working knowledge of indexing in Matlab would be to for Alice to reorder the columns of her plaintext matrix before reading off the columns to form the ciphertext.

Learning Objectives

After the lesson, students should be able to

- Encrypt and decrypt small messages using the standard transposition cipher without using reference materials
- Define matrices and vectors and explain the connection between matrices and vectors and the representation of plaintext and ciphertext in the transposition cipher
- Articulate the functionality of the `reshape` command
- Encrypt and decrypt large messages using the Matlab implementation of the standard transposition cipher which they have written
- Explain how frequency analysis indicates a transposition encryption
- Cryptanalyze transposition ciphers using a brute force `for` loop attack using code they have written

C.4 Teaching Note 2.2: Caesar Ciphers

Caesar ciphers are some of the most common cryptosystems in popular culture. Students may be familiar with old-fashioned "decoder rings" and these toys could provide a natural and informative in-class activity. What may be more surprising is the historical context of these codes, including primary source material detailing Julius Caesar's use of the cryptosystem. This provides another touchstone for the idea that many cryptosystems are alphabet agnostic.

In addition to the cryptographic content, this lesson is meant provide an introduction to the concepts of modular arithmetic and groups. In particular the notion of additive inverses is integral for Caesar decryption. Moreover, the notion of additive inverses naturally motivates the discussion of multiplicative inverses necessary in affine cryptosystems. Introducing groups in one lesson and rings in another breaks up some of the more technical content of the respective lessons in order to allow students who may not be familiar with discrete structures more time to adequately internalize the material.

The studio problems associated with this lesson (Section B.4 on page 103) require that students implement the Caesar cryptosystem in code. (It is also expected that students will be able to perform encryption/decryption of simple messages by hand.) The `for` loop implementation of the brute force attack may be challenging for students. Section 1.2 provides a more complete discussion of `for` loops and their syntax. On the theoretical side, there several studio problems to help students consider the idea of groups under multiplication, which motivates the discussion of multiplicative inverses in the affine cryptosystems discussed in Section 2.3. In particular, students should at least begin to realize that the additive identity 0 cannot have an inverse and that some nonzero elements may fail to have multiplicative inverses.

Learning Objectives

After the lesson, students should be able to

- Encrypt and decrypt small messages using a Caesar cipher without using reference materials
- Perform arithmetic modulo n without using reference materials, including operations involving additive inverses
- Encrypt and decrypt large messages using the Matlab implementation of a Caesar cipher which they have written
- Cryptanalyze Caesar ciphers using a brute force `for` loop attack using code they have written
- Articulate the group axioms and relate the inverse and identity axioms to Caesar cryptosystems

C.5 Teaching Note 2.3: Affine Ciphers

Affine ciphers are a natural extension of Caesar ciphers in the quest for larger and larger key spaces. As with Caesar ciphers, the encoding rule is relatively simple. It is the difficulty of decoding that sets the two approaches apart. Affine ciphers for this reason are an excellent introduction to the idea that in cryptography it is often easy to perform an action, but it can often be difficult to *prove* that this action always does what you intend. In the context of affine ciphers, the decoder relies on the idea that we can "undo" modular multiplication. Students' mathematical intuition gained from dealing with real numbers indicates that such a reversal should always be possible. This leads to a natural motivation of a suitable definition of an inverse, how we could know *a priori* whether a particular element will have a multiplicative inverse modulo n, and, ultimately, to investigation of units and zero divisors.

The basics of several proof strategies are deployed in the investigation of the structure of units and zero divisors; these techniques might prove useful if other topics, particularly those from modern cryptography, will be covered later in the course. Moreover, the concept of the multiplicative group of a ring is one that recurs throughout modern cryptography. The exact extent to which these ideas are introduced mathematically can be easily tailored to a given class.

As with Caesar ciphers, affine ciphers have a relatively simple encoder/decoder structure which should help students with limited programming experience practice how to structure functions. Common mistakes include writing the modular inverse of ℓ in Matlab as `ell^{-1}`. The code bundled with these notes includes a function `modInv(x,n)` which produces the inverse of x modulo n. Students with a strong mathematical background may enjoy rederiving the formulation used `modInv` from Bézout's identity.

Learning Objectives

After the lesson, students should be able to

- Articulate the definitions of and distinction between the zero divisors and units of a ring
- Encrypt and decrypt small messages using an affine cipher without using reference materials
- Encrypt and decrypt large messages using a Matlab implementation of the affine cryptosystem which they have written
- Cryptanalyze affine ciphers with a brute force attack using code they have written

C.6 Teaching Note 2.4: Polyalphabetic Ciphers

Polyalphabetic ciphers are a natural bridge between classical and modern cryptosystems. The many variations on a common theme allow for students to explore the implications of small changes in encryption to security and ease of use. The lesson serves as an excellent capstone to classical cryptosystems.

Vigenère ciphers with relatively simple keys are little more than natural extensions of Caesar ciphers. Students can often reinvent running key ciphers themselves in a quest to expand the key space. A brief discussion of how the properties of large blocks of English text can be exploited to break running key ciphers motivates the development of one-time pads. This brings students to the edge of the art, in the sense that one-time pads are "unbreakable" if implemented properly. This notion of unbreakability can be contrasted with the discrete logarithm problem and the integer factoring at the heart of many modern crypto graphic techniques; these problems are thought to be difficult but may in fact be tractable.

Vigenère ciphers likely present the most difficult cryptanalysis students will have yet seen. If students have encountered the frequentist attack on Caesar ciphers, Friedman's method of coincidences will take them the rest of the way. While strategy can be presented effectively without too much formal mathematics, more advanced students can formally derive the associated probabilities. The increasing difficulty of cryptanalysis as one moves from Vigenère ciphers to running key ciphers is a good introduction to the idea that some seemingly simple cryptosystems may be very difficult to attack formulaically. A discussion of one-time pads should send this message home.

The studio problems associated with this lesson require students to implement a general polyalphabetic encrypter and decrypter. There is a teachable point here: a small increase in effort towards generality can lead to a large increase in productivity.

Learning Objectives

After the lesson, students should be able to

- Articulate the process of Vigenère encryption and decryption
- Encrypt and decrypt Vigenère, running key, and one-time pad ciphers using code they have written
- Articulate the process of Vigenère cryptanalysis, including Friedman analysis and frequency analysis
- Perform Vigenère cryptanalysis using a combination of code they have written and guess-and-check
- Articulate the difference between the sizes of different polyalphabetic key sizes

C.7 Teaching Note 3.1: Diffie-Hellman

The Diffie-Hellman key exchange algorithm may be the first time that students see the inner workings of a modern cryptographic protocol. Perhaps more importantly, the topic often leads to a series of "firsts" for many students. For most, the topic will be the first time they see interesting properties of groups, like primitive roots, in action; even a student with a strong group-theoretic background will likely not have seen such an application. The discrete logarithm problem may be some students' first encounter with the state of the art of mathematics. In particular, applied students may not realize there are things in mathematics that no one knows. Modular exponentiation of large numbers requires an introduction to the digital representations of integers and the shortcomings of these representations. Finally, students who have been exposed to classical cryptanalysis might be somewhat shocked to see just how easily even cryptosystems resting on strong mathematical footing can be compromised, for instance through a simple man-in-the-middle attack.

Students may also find the historical context of the problem interesting. In the early 1970s, several different researchers independently converged on the idea of using discrete logarithms to encrypt data without the used of a shared private key. While Diffie and Hellman ultimately had their names ascribed to the key exchange technique, British researchers Ellis, Cocks, and Williamson, working in secret at the Government Communications Headquarters in the United Kingdom actually developed the algorithm first. Their work remained classified.

The topic leads naturally into a series of mathematically related applications, including the El Gamal cryptosystem and the Digital Signature Algorithm (DSA). (The latter also naturally induces the concept of hash functions.) Strong students can be guided towards several challenges. For instance, an investigation into and implementation of efficient modular exponentiation by squaring is perfectly suited for an advanced undergraduate with some prior coding experience. For even more advanced students, an investigation into the trial multiplication and baby-step-giant-step discrete logarithm algorithms might be interesting.

Learning Objectives

After the lesson, students should be able to

- Explain the sequences of transactions that take place in Diffie-Hellman key exchange
- Define primitive roots and relate them to Diffie-Hellman key exchange
- Articulate the nature and difficulty of the discrete logarithm problem
- Compute the public keys and resulting shared secret in the Diffie-Hellman scheme using code they have written

C.8 Teaching Note 3.2: RSA

The Rivest-Shamir-Adelman (RSA) algorithm is one of the mainstays of modern cryptography. As a pedagogical tool, it is useful for introducing a number of important concepts, and these ideas can be put in context by comparing RSA to other modern cryptographic schemes. Unlike Diffie-Hellman, RSA can be used natively to exchange messages. Whereas classical cryptosystems encrypt plaintext English, RSA encrypts numeric messages. This naturally leads to a discussion of how modern computers store information. Depending on the audience, the presentation could be limited to a scheme for mapping between binary string and decimal representations. A more involved discussion could involve the ASCII alphabet and/or the fundamentals of binary arithmetic. In any case, students can gain valuable experience with some of the underpinnings of modern computing.

Like Diffie-Hellman, RSA relies on the commonly accepted difficulty of a particular arithmetic operation, here the factorization of products of large primes. This problem is embedded just below the surface in the RSA scheme, and to fully investigate its relevance, Euler's totient function and the concept of mulitplicative order must be introduced. Both have deep relationships to concepts students may have already seen in other cryptosystems. In particular, students can explore the connections between primitivity (as seen in Diffie-Hellman) and the maximum order of an element as well as the connections between Euler's totient function and the collection of units in a given ring (as seen in Affine Ciphers). Both should help solidify previous experience. The extent to which various connections are fleshed out through proof can be widely varied to fit the audience. For instance, the idea that an element is a unit if and only if it has finite order can be either proved during lecture, presented as a result without proof, or left as an exercise depending on students' familiarity with the fundamentals of proofs. (Some of these results are included in the studio problems associated with this lesson.) In particular, these notes present Lagrange's Theorem as a result, since the notion of cosets is a large piece of machinery to introduce for an intermediate result.

Learning Objectives

After the lesson, students should be able to

- Define Euler's totient function and articulate its importance in the RSA algorithm
- Define the multiplicative order of an element of $\mathbb{Z}_n$ and articulate its importance in the RSA algorithm
- Articulate the nature and difficulty of the integer factorization problem
- Encrypt given plaintext and decrypt given ciphertext in the RSA cryptosystem using code they have written

C.9 Teaching Note 3.3: Cryptographic Hashes

Cryptographic hashes are an interesting topic to teach in that their design and implementation are likely too complicated and esoteric for a general undergraduate course, while their applications are numerous and powerful. Typically, students can examine the details of a cryptographic technique and come away with a deeper understanding of general techniques that will help them in the field. The same is arguably true for cryptographic hashes, though the start-up cost is much higher. Accordingly, this section examines the basic ideas behind cryptographic hashes, including desirable properties and the relationships between these properties and other cryptographic ideas, while completely disregarding the actual methodology through which hashes are computed.

In some ways, cryptographic hashes provide a very natural bridge between "theoretical" cryptography and "applied" cryptography, in the sense that cryptographic systems that are theoretical secure can sometimes fail in a real world context. A chain of examples that is laid out in the studio problems associated with this section is password storage. Most students will recognize that storing passwords in plaintext is a bad idea, but most students will think that encrypting passwords is a good idea. But academic assumptions often ignore on-the-ground facts, here namely that an insider with access to the private key could easily steal the plaintext passwords. True to form in this applied vein, hashing the passwords does not solve the problem entirely; it simply makes breaching security more difficult. The addition of salts increases the difficulty further. This progression is a good one for students to see, especially in the latter part of a course. One should also highlight the fact that the solutions our instincts tell us should be good are often very bad, and that rubber-meets-the-road cryptography should be left to professionals whenever possible.

The Matlab code bundled with these notes includes a hash function `sha0`. This is a truncated version of the defunct `sha1` hash function which returns an integer rather than a bit or hex string. The maximum range of the integer is chosen to avoid overflow of Matlab's default double precision. Moreover, this decreased output range should allow students to generate collisions reliably; this could be leveraged in a potentially interesting problem in which students generate a fixed number of collisions and determine to what extent the birthday bound, which is detailed in this section, actually holds.

Learning Objectives

After the lesson, students should be able to

- Define both pre-image resistance and collision resistance
- Articulate the importance of both pre-image resistance and collision resistance in a given application of cryptographic hashes
- Explain the birthday paradox and how it relates to collisions in cryptographic hashes

C.10 Teaching Note 3.4: Digital Signatures

Digital signatures dovetail very nicely with cryptographic hashes, in the sense that they give a concrete and rigorously mathematical application of the general concept of hashing. This section presents two: an RSA-based signatures scheme and the DLP-based ElGamal signature scheme. In each, the key concept is that an effective signature should include both information about the hash of the message and the private key information. Understanding how and why these pieces of information are included in a given digital signature is integral towards understanding the concept as a whole.

While digital signatures are often very important, the choice as to whether to encrypt the message itself before transmission over the channel is left to the system's designer. Accordingly, these notes provide an example of each. This could naturally lead to a discussion as to when encryption of the message would be a good thing (*e.g.*, sensitive data, contracts), and when it might be computationally wasteful (*e.g.*, downloading a game in an app store, downloading an update on a gaming console). These notions of some data being "safe" for sharing could be compared and contrasted to recent calls for encryption of all internet traffic, regardless of sensitivity.

The ephemeral value k in the ElGamal signature scheme leads to a great example of how seemingly strong cryptographic protocols can fail if implemented incorrectly. If 2010, the hacking group `fail0verflow` gained access to Sony Entertainment's private DSA key in part by exploiting Sony's decision not to generate a fresh value of k for each signature. While DSA is more complicated than ElGamal, the fundamentals are similar enough that a similar attack can be brought to bear in the more restricted context; the notes for this section include a worked example. One natural extension of this exercise would be to provide students with message-signature pairs in a given ElGamal signature setup and ask if the information provided was sufficient to determine whether the value of k was being reused and, if so, what this value of k was.

Learning Objectives

After the lesson, students should be able to

- Explain the importance of digital signatures in a wider cryptographic context
- Explain how hash and private information is integrated into RSA-based signatures
- Compute and verify RSA-based signatures using code they have written
- Explain how hash and private key information is integrated into DLP-based signatures
- Compute and verify DLP-based signatures using code they have written

C.11 Teaching Note 3.5: Zero-Knowledge Proofs

A discussion of zero-knowledge proofs at the end of a undergraduate cryptography course has several advantages. First, it offers a "break" from the more quantitatively demanding methods of modern cryptography. Second, it presents clear connections between probability and cryptography. Until this point, many students will have seen very algebraically beautiful and perfectly rigid cryptographic techniques. Zero-knowledge proofs hint that there is a wider world to explore, and indeed many of the more powerful tools from cryptography and coding theory have a probabilistic flavor, *e.g.* Shannon's original theorems, low density parity check codes, *etc.* Third, students may have repeatedly encountered the idea that it is difficult to digitally verify a person's identity. Zero-knowledge proofs offer one solution. This connection also allows for a discussion of trusted third parties and the importance of the chain of trust in modern digital communications.

The lesson begins with a non-mathematical example of zero-knowledge proofs. This example is even simpler than the traditional "magic cave" example, though it does not contain the idea of "committing" as featured in many zero-knowledge proof systems. The lesson goes on to discuss two mathematical examples, Schnorr authentication and Fiat-Feige-Shamir authentication. The former offers nice concoctions to the discrete logarithm problem. The latter allows for a discussion of computational efficiency, in that the FFS scheme allows for many verifications to be performed with the same exchange. In classes with sufficient engineering and/or computer science background, the FFS scheme demonstrates the need to understand the latency and computational complexity considerations in a given application.

Many of the other mathematical examples of zero-knowledge proofs are quite involved, and so the studio problems for this section focus on the concepts rather than the mathematics. In particular, students are repeatedly asked to apply the concepts of completeness, soundness, and zero-knowledge to proposed schemes.

Learning Objectives

After the lesson, students should be able to

- Articulate the roles of the prover and verifier in a zero-knowledge proof system
- Explain the importance of completeness, soundness, and zero-knowledge in general zero-knowledge proof systems
- Apply the concepts of completeness, soundness, and zero-knowledge to a given application
- Design zero-knowledge proof solutions