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.

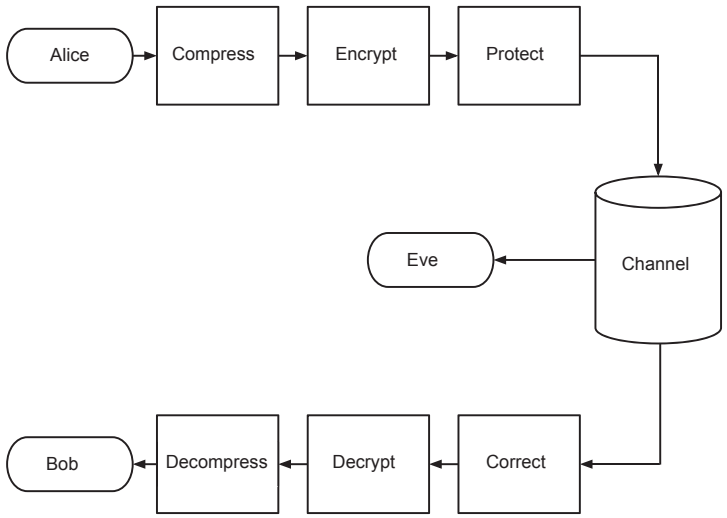

Figure 1.1: General communication system with Alice sending a message to Bob through a public channel monitored by Eve.

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
```