CS 198 Codebreaking at Cal
Spring 2023 Course Notes

Note 1

Introduction to Classical Cryptography

Like every other science, cryptography has a rich history beginning all the way back with the Roman Empire. The Romans developed a very simple method of hiding messages from those who didn't know the secret key (we will learn about this shortly). However, this system is laughably weak in the era of computers. This pre-computer cryptography is often called **classical cryptography**, which largely consists of pen-and-paper encryption schemes with minimal security. These systems often rely on attackers to not know how the system works rather than inherently strong security. Steganography (the art of hiding messages in innocuous images/text) is another example of classical cryptography, which we will learn about in this note as well!

The word "codebreaking" is largely associated with the analysis and attack of classical cryptosystems, since they are often called "codes" rather than cryptosystems. Note that this class will largely focus on **modern cryptography** despite the name, though we retain the focus on exploiting systems.

Substitution Ciphers

The most basic type of cipher is the *substitution cipher*, where the user translates their message (plaintext) from one alphabet to a permutation of that alphabet. To review:

Definition 1: Cipher

A cipher is a function that encrypts some message (**plaintext**) into an unreadable **ciphertext** using a **key**. This ciphertext can be decrypted back into plaintext later by someone who knows the key.

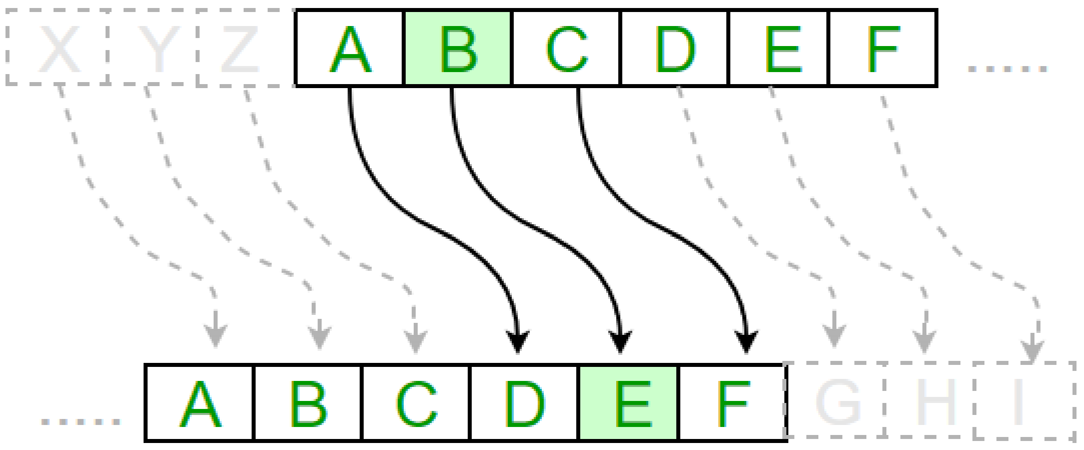
For example:

Plaintext:	aaaa
Key:	1
Ciphertext:	bbbb

Which can be translated to numbers as such (starting at $a = 0$ ...):

Plaintext:	0000
Key:	1
Ciphertext:	1111

As you can see, this cipher shifts all letters over by some number. This cipher is called a **Caesar cipher**, one of the earliest known cryptosystems. Visually, we can see it like this (for a key of 3) [1]:



We can formally model this as a function E_n:

$$A := \{a, b, ...z\}$$
$$E_n : A \to A$$
$$(\forall x \in A)(D_n(E_n(x)) = x)$$

In effect, we define a function E_n that operates on alphabet A, and whose inverse exists for all elements in A. This function is the *encryption function*, and its inverse the *decrypting function* (D_n). For addition-based substitution ciphers like this, the decryption is simply subtracting the key.

Notably, this inverse must exist mod 26, as our alphabet has 26 letters. If we were to start at "z" (25) and add 1, we would "wrap around" to "a" (0). This relationship is modeled as follows:

$$E_n(x) = x + n \pmod{26}$$
$$D_n(x) = x - n \pmod{26}$$

Since this cipher is additive, we do not have to worry about the existence of a multiplicative inverse (however, we will in later topics using modular arithmetic).

Exercise 1.1: Decrypt NV ILHYZ using a shift of 7.

Polyalphabetic Ciphers

So far, we have seen a monoalphabetic cipher (Caesar), which employs a single alphabet. However, there exists ciphers that utilize multiple substitution alphabets.

[1] Credit https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/

Definition 2: Repeating key

A **repeating key** is a way of extending a key that is shorter than the plaintext to be the same length as the plaintext. For example, the plaintext "hello" and key "bc" would end up with a repeated key of "bcbcb".

The most famous one, named the *Vigenère cipher*, is a straightforward extension of the Caesar cipher using a repeating key.

Plaintext:	aaaa
Key:	ab
Ciphertext:	abab

Using our numbering system and extending the key to fit the full ciphertext:

Plaintext:	0000
Key:	0101
Ciphertext:	0101

Despite the plaintext being all the same letter, the ciphertext is different! This provides a minute amount more security than a Caesar cipher, but as you will see later, both are easily broken with some statistical analysis.

Formally:

$$\begin{aligned} n &= \text{length of the key} \\ E_i(x) &:= x_i + k_i \pmod{26} \\ k_i &= (i \pmod{n})\text{-th digit of the key} \\ x_i &= i\text{-th digit of the plaintext} \end{aligned}$$

Thus,

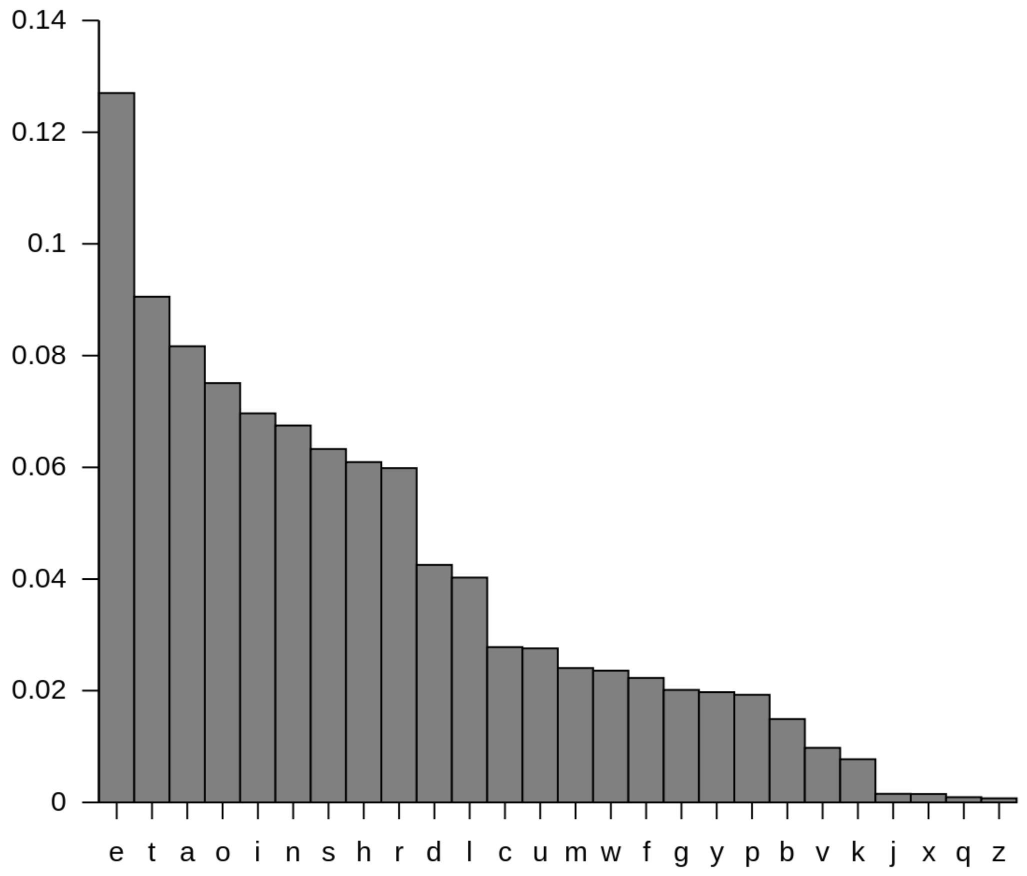
$$E(x) := E_1(x) \| E_2(x) \| \dots \| E_n(x)$$

The decryption function is defined similarly, just subtracting the key instead. These structures work for any arbitrary alphabet and encryption function (presuming that the inverse always exists). You could represent your data in binary, hexadecimal, or any base with no problems. Encrypting and decrypting is as simple as addition and subtraction modulo some number. For this reason, such ciphers dominated the early years of cryptography, as they provided easy and quick security. However, as we will see in the next page, the security is basically nonexistent.

Exercise 1.2: Decrypt "fw fmohb-sai-r oqrbyeqhqrrkg" using key "denero".

Frequency Analysis

Unfortunately for our newly constructed ciphers, there exists a very powerful tool at our disposal to detect what the shift of a monoalphabetic cipher is: *frequency analysis*. Simply put, English (and other languages) use some letters more than others. Over a large set of numbers, the distribution will converge to be something like this:



Using this, we can construct a function to "score" any given blurb of English text by comparing it to this distribution. If the frequencies of the given text are close to the frequencies of this distribution, we give it a high score, meaning it is likely to be English.

However, small texts often have very high variance, meaning they won't necessarily follow the distribution even if it is valid English. For example, "fun" is a valid word but virtually indistinguishable from gibberish using this method since it is so short. The Law of Large Numbers implies that as the length of this hypothetical English text gets longer, the distribution will converge to the one pictured above. Thus, it will work for *most* examples.

Definition 3: Law of Large Numbers

The **Law of Large Numbers**, or LLN for short, states that as the number of experiments approaches infinity, the average of the results will converge to the expected value.

Now, how to implement this in mathematically? There are various methods of frequency analysis, with more advanced ones taking into account subparts of words like "-ing", but for now we will stick to simply comparing the frequencies of each individual letter. For those who have taken some statistics classes (or Data 8), the concept of *Total Variation Distance* may be familiar. TVD is a method of calculating how "different" two distributions are by summing the absolute value of the differences between each corresponding entry.

Definition 4: Total Variation Distance

The **Total Variation Distance**, or TVD for short, is a method for measuring the "distance" between two categorical distributons. The function $f(x,y)$for finding the TVD is as follows for distributions x,y with n categories:

$$f(x,y) = \frac{1}{2}\sum_{i=0}^{n} | x_i - y_i |$$

Given base frequency values $F_0 \ldots F_{25}$ and ciphertext frequency values $C_0 \ldots C_{25}$, where F_0 represents the frequency of A, we wish to find:

$$\min_k \frac{1}{2}\sum_{i=0}^{25} | (C_{i+k} - F_i) |$$

Informally, we find the offset with the smallest TVD, which translates to the offset giving us the most English-like decrypted text. This offset is our best guess at the key.

Polyalphabetic Frequency Analysis

While this basic form of frequency analysis works well for monoalphabetic ciphers like Caesar's cipher, it does not work well with polyalphabetic ones. This is because the multiple alphabets have different frequency charts, which overlap and eventually form a uniform distribution.

However, **if** we are able to discern the key length (say, k), we can split the cipher into k different monoalphabetic ciphers to crack individually. For example, a Vigenère cipher with key length 2, ciphertext "BCBC" can be split into "BB" and "CC" individual, monoalphabetic ciphers.

Once solved via frequency analysis, we take the offset and add it back into the overall key. For this example, the offsets are 1 and 2, so the key is 12 = "BC".

The trouble lies in determining the key size. For a very long message of length n, attempting to try every different key size results in $O(n^2)$ operations. In the grand scheme of cryptography, this polynomial-time algorithm is actually very tractable. However, we can do better with statistical analysis.

Definition 5: Big-O Notation

Big-O notation is a way of analying the complexity of an algorithm. Informally put, a algorithm with input size n taking $O(n^2)$ time means it will take n^2 operations as $n \rightarrow \inf$. "Tractable" means computationally feasible to run in a reasonable amount of time.

The **Kasiski test** utilizes repetitions of characters to identify the most likely key size. For example, take the following encryption:

Plaintext:	Cryptography isnt cryptocurrency
Key:	abcd
Ciphertext:	**Csastp**iuaqjb itpw **csastp**exrsgqcz

Notice the bolded sections – these letters are the same, meaning that it is very likely they were the same plaintext encoded by the *same key*. We can use this to our advantage by recording how far apart these two sections are, to try and find a key length that would "repeat" at exactly that length. In this case, measuring from the 'o' to the next 'o' in crypto, we see they are 16 characters apart. 4 divides 16, and thus is far more likely to be the key length than other distances. Of course, so do 2 and 8 – but as your ciphertext gets longer and longer, all these occurrences of duplicate substrings will narrow down the choice quite considerably.

Open-Ended Question: Is there a way to modify this cipher to be more resistant to Kasiski tests?

Steganography

Even in theoretical cryptography, there are situations in which we don't have a reliable channel to send information through. Vigilant authorities might block the transmission of any messages that appear to have a secret message, or are encrypted. Adversaries might even have a secret method to break our encryption, so brazenly sending encrypted messages would be insecure. Normal circumstances usually do not call for such heavy constraints, but there are specific situations in which it is paramount to communicate regardless of these adversaries.

Before we consider how encryption might help us here, we might want to consider a different approach entirely. Other methods of keeping data secret do exist, most of which attempt to 'encode' data where attackers are unlikely to look without knowing the secret. These methods are often called **steganography**.

In fiction, these would often appear as a message hidden in a letter, often as the first letter in each sentence comprising the entire message[2]. Some spies would even hide information in crossword puzzles!

Often times, however, we want to use more modern methods of transmitting secret messages using steganography. Still, we need to keep up the appearance of a totally normal message, which can be tricky when modifying an image. Key to this plan is the nature of computer-based graphics, which provide a large amount of flexibility for encoding information. If we only modify the last bit of the bit representation of a pixel, the overall image appears unchanged.

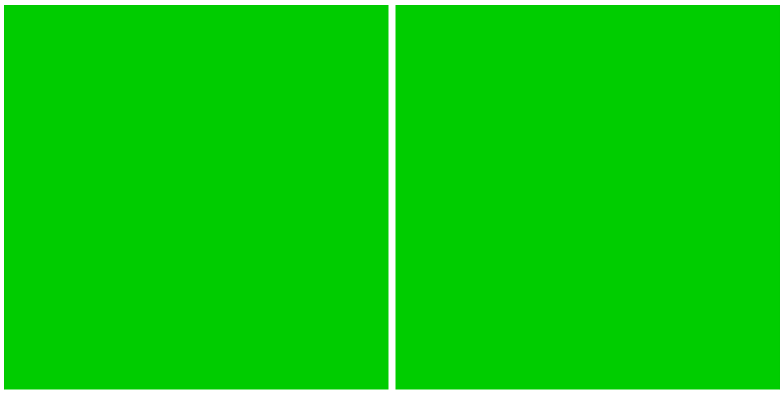
Digital Images

To explore how this is possible, we must first look at how images are stored within computers. At the most basic level, we need to represent each pixel as a combination of red, green, and

[2] If you haven't found the hidden message yet, I recommend reading this line again!

blue. Computers do this by assigning each color a value from 0 to 255, representing (roughly) the intensity of that color within the pixel. For example, (255, 0, 0) is all red, (0, 0, 0) is white, and (255, 255, 255) is black.

As a result, we can represent a wide range of colors as a combination of these three (a total of $2^{24} = 256^3 = 16777216$). Note that each color is represented by 8 bits ($2^8 = 256$) as a binary string. The nature of binary makes it so modifications to the least significant bit only change the overall number by at most 1. The change in color intensity is nearly impossible to detect by the naked eye: take a look at the following (very large) pixels:

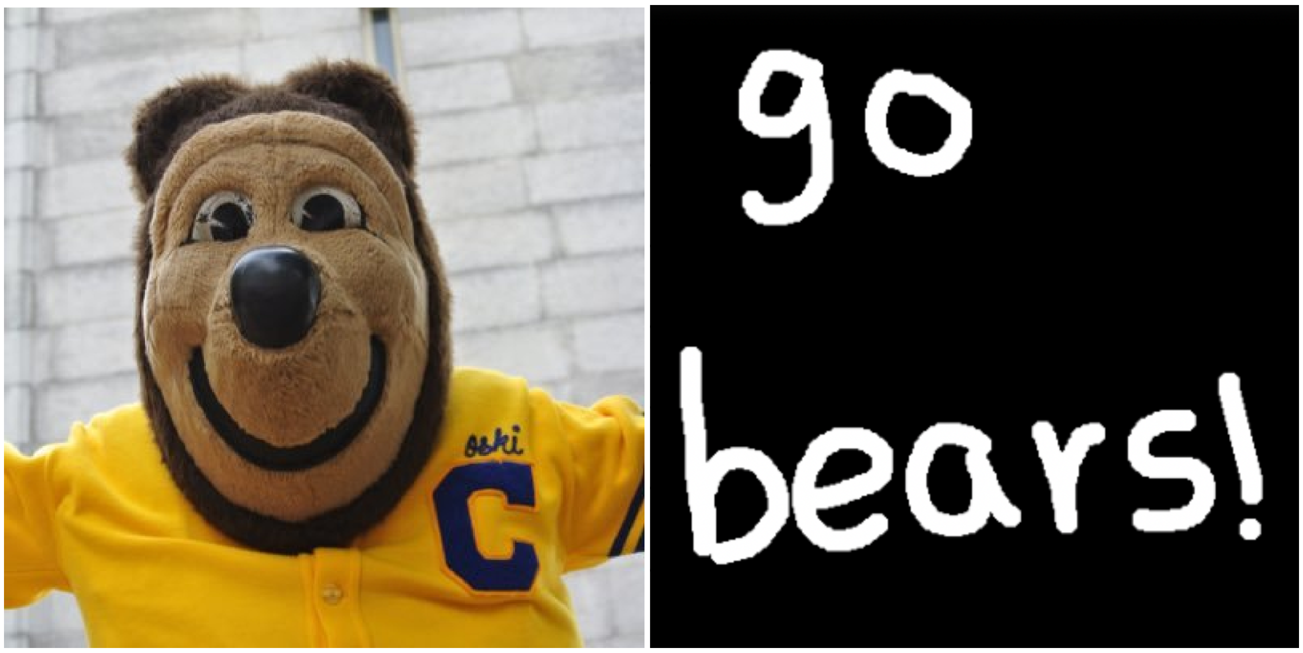


One is [0,204,0] and the other is [0,205,0]. Humans simply cannot tell the difference at a glance (especially when its one tiny pixel alongside millions of others).

Encoding Information in Images

In order to encode a message, we then need a scheme for changing the values of the last bit to something specific. We could always just denote a normal bitstring message as the result of concatenation every last bit of each pixel. This would work for text-based messages.

Image-based messages can be done in a similar fashion. If the last pixel of the image is 0, we set the message pixel to white, and black otherwise. This forms a new, black-and-white image that can convey a message.



The image of Oski on the left encodes the image of "go bears!" on the right.

Crucially, however, images provide a far better **expansion ratio**, or how much more "normal message" is needed to hide the secret message. The expansion ratio for encoding a message as the first letter of every sentence is abysmal, probably upwards of 1 character of secret message to every 50 of normal message. For images, each pixel can transmit multiple bits of information. In a 1024x1024 image, that's hundreds of kilobytes of data!

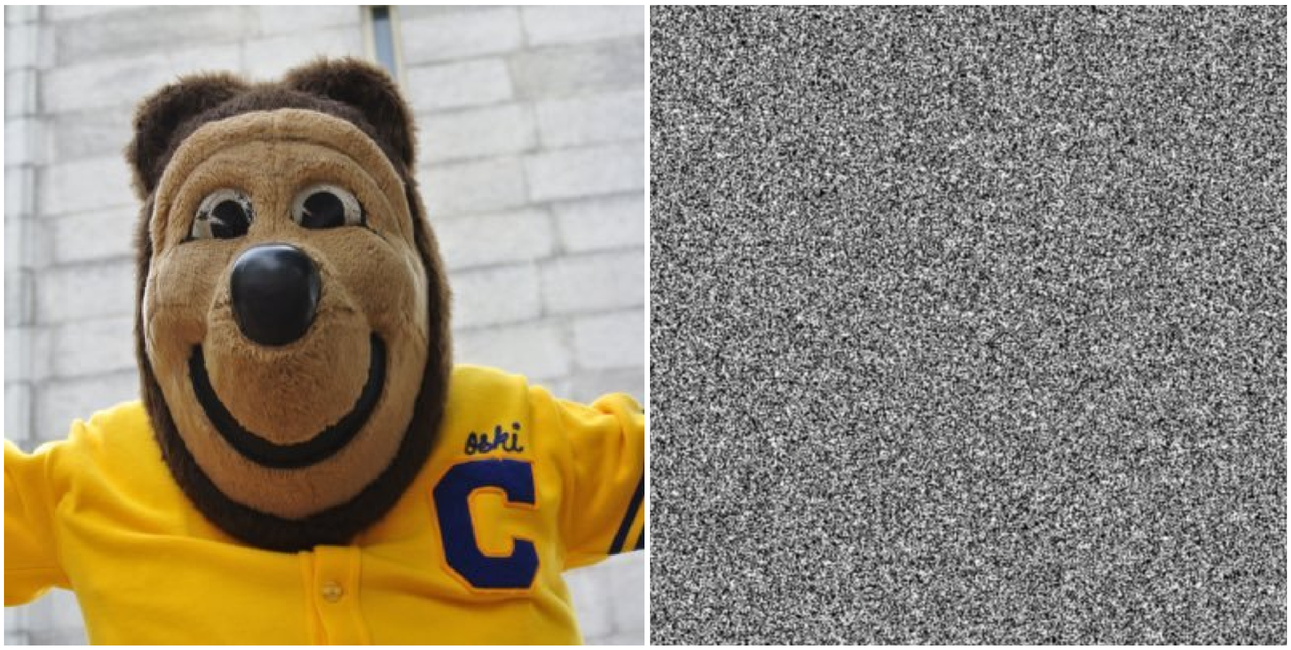
Steganalysis

Methods do exist to try and detect when steganography has occurred, but it is difficult to do so without knowing where to look. The situation is also highly dependent on whether the stegoanalyst has access to more than one steganographically-modified images, or if they even know steganography has occurred at all. Overall, steganalysis is considered incredibly difficult.

In certain cases, it is possible to compare the expected proportions of data to the observed proportions. If the last bits of a large image were made of 80% 0s, something is probably fishy – it is exceedingly unlikely for that to happen by chance except in rare circumstances (like a lot of whitespace in the background).

Content Threat Removal

While it may not be easy to detect whether a message is steganography, it is easy to mess with the outcome. Since we have seen that modifying the last bits of an image has negigible effect on the overall image, if we randomly scramble them, it should have no real effect either. Randomly scrambling the last bits of every image would also ruin any messages send by steganography, so there is little downside to doing so. Of course, this is dependent on knowing the scheme used for steganography.



This is the result of running our decoding algorithm on a scrambled image, while the original still looks fine.

Contributors:

- Ryan Cottone