Discussion

We'll conclude with some basic advice for programmers [illegible] to use cryptography in their software. Rather than trying [illegible] own crypto functions, keep in mind it's best to use [illegible] braries. Libraries are good for you—there are many ways [illegible] tems can fail, and experts have thought about defending [illegible] many potential attacks. Don't be a cowboy programmer—[illegible] libraries.

Links

[Signal is a secure messaging app for mobile]
`https://whispersystems.org/`

[Public-key cryptography general concepts]
`https://en.wikipedia.org/wiki/Public-key_cryptogra[illegible]`

Exercises

E7.14 Alice wants to send the message $\vec{m} = 0110\ 1000$ [illegible] Bob. They have pre-shared the secret key $\vec{k} = 1010\ 0111$ [illegible] Compute the ciphertext $\vec{c} = \text{Enc}(\vec{m}, \vec{k}) = \vec{m} \oplus \vec{k}$ that Alice [illegible] Bob. Verify that Bob will obtain the correct message after [illegible]

7.10 Error-correcting codes

The raw information-carrying capacity of a DVD is roughly [illegible] which is about 20% more than the 4.7GB of data that your [illegible] will let you write to it. Why this overhead? Are DVD manu[illegible] trying to cheat you? Actually, they're looking out for you; [illegible] space is required for the *error-correcting code* that is applied [illegible] data before writing it to the disk. Without the error-correcting [illegible] even the tiniest scratch on the surface of the disk would [illegible] disk unreadable, destroying your precious data. In this section, [illegible] learn how error-correcting codes work.

Error-correcting codes play an essential part in the [illegible] the transmission, and the processing of digital information. [illegible] the slightest change to a computer program will make it [illegible] computer programs simply don't like it when you fiddle [illegible] bits. Crashing was the norm back in the 1940s as illustrated [illegible] quote:

> "Two weekends in a row I came in and found that all my stuff had been dumped and nothing was done. I was really annoyed because I wanted

those answers and two weekends had been lost. And so I said, Dammit, **if the machine can detect an error, why can't it locate the position of the error and correct it?"**

—Richard Hamming

Richard Hamming was a researcher at Bell in the 1940s. He ran into the problem of digital data corruption, and decided to do something to fix it. As a solution, he figured out a clever way to encode k bits of information into n bits of storage, such that it's possible to recover the information even if some errors occurred on the storage medium. An *error-correcting code* is a mathematical strategy for defending against erasures and errors. Hamming's invention of error-correcting codes became a prerequisite for the modern age of computing—after all, reliable computation is much more useful than unreliable computation.

Definitions

An *error-correcting code* is a prescription for encoding *binary* information. Recall that bits are elements of the binary field, $\mathbb{F}_2 = \{0,1\}$. A bitstring of length n is an n-dimensional vector of bits $\vec{v} \in \{0,1\}^n$. For example, 0010 is a bitstring of length 4.

We use several parameters to characterize error-correcting codes:

- k: the size, or length, of the messages for the code.
- $\vec{x}_i \in \{0,1\}^k$: a *message*. Any bitstring of length k is a valid message.
- n: the size of the codewords in the code.
- $\vec{c}_i \in \{0,1\}^n$: the *codeword* that corresponds to message $\vec{x}_i$.
- A *code* consists of 2^k codewords $\{\vec{c}_1, \vec{c}_2, \ldots\}$, one for each of the possible messages $\{\vec{x}_1, \vec{x}_2, \ldots\}$.
- $d(\vec{c}_i, \vec{c}_j)$: the *Hamming distance* between codewords $\vec{c}_i$ and $\vec{c}_j$.
- An (n,k,d) code is a procedure for encoding messages into codewords; $\text{Enc} : \{0,1\}^k \to \{0,1\}^n$, which guarantees the *minimum distance* between any two codewords is at least d.

The *Hamming distance* between two bitstrings $\vec{x}, \vec{y} \in \{0,1\}^n$ counts the number of bits where the two bitstrings differ:

$$d(\vec{x},\vec{y}) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \delta(x_i, y_i), \quad \text{where } \delta(x_i,y_i) = \begin{cases} 0 & \text{if } x_i = y_i, \\ 1 & \text{if } x_i \neq y_i. \end{cases}$$

Intuitively, the Hamming distance between two bitstrings measures the minimum number of substitutions required to transform one bitstring into the other. For example, the Hamming distance between codewords $\vec{c}_1 = 0010$ and $\vec{c}_2 = 0101$ is $d(\vec{c}_1, \vec{c}_2) = 3$, because it takes three substitutions (also called *bit flips*) to convert $\vec{c}_1$ to $\vec{c}_2$ or vice versa.

An (n, k, d) code is defined by a function $\text{Enc} : \{0,1\}^k \to \{0,1\}^n$ that encodes messages $\vec{x}_i \in \{0,1\}^k$ into codewords $\vec{c}_i \in \{0,1\}^n$. Usually the encoding procedure Enc is paired with a decoding procedure, $\text{Dec} : \{0,1\}^n \to \{0,1\}^k$, which recovers messages from (possibly corrupted) codewords.

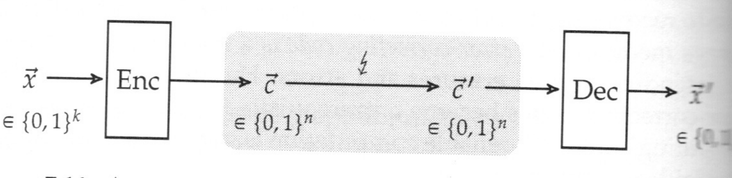

Figure 7.14: An error-correcting scheme using the encoding function Enc and the decoding function Dec to protect against the effect of noise (denoted $\lightning$). Each message $\vec{x}$ is encoded into a codeword $\vec{c}$. The codeword $\vec{c}$ is transmitted through a *noisy channel* that can corrupt the codeword by transforming it into another bitstring $\vec{c}\,'$. The decoding function Dec looks for a valid codeword $\vec{c}$ that is close in Hamming distance to $\vec{c}\,'$. If the protocol is successful, the decoded message will match the transmitted message $\vec{x}\,' = \vec{x}$ despite the noise ($\lightning$).

Linear codes

A code is *linear* if its encoding function Enc is a linear transformation:

$$\text{Enc}(\vec{x}_i + \vec{x}_j) = \text{Enc}(\vec{x}_i) + \text{Enc}(\vec{x}_j), \text{ for all messages } \vec{x}_i, \vec{x}_j.$$

An (n, k, d) linear code encodes k-bit messages into n-bit codewords with minimum inter-codeword distance d. Linear codes are interesting because their encoding function Enc can be implemented as a matrix multiplication. We use the following terms when defining linear codes as matrices:

- $G \in \mathbb{F}_2^{k \times n}$: the *generating matrix* of the code. Each codeword $\vec{c}_i$ is produced by multiplying the message $\vec{x}_i$ by G from the right:

$$\text{Enc}(\vec{x}_i) = \vec{c}_i = \vec{x}_i G.$$

- $\mathcal{R}(G)$: the row space of the generator matrix is called the *code space*. We say a codeword $\vec{c}$ is valid if $\vec{c} \in \mathcal{R}(G)$, which means there exists some message $\vec{x} \in \{0,1\}^k$ such that $\vec{x}G = \vec{c}$.
- $H \in \mathbb{F}_2^{(n-k) \times n}$: the *parity check matrix* of the code. The *syndrome* vector $\vec{s}$ of any bitstring $\vec{c}\,'$ is obtained by multiplying $\vec{c}\,'^{\mathsf{T}}$ by H

from the left:

$$\vec{s} = H\vec{c}'^{\mathsf{T}}.$$

If $\vec{c}'$ is a valid codeword (no error occurred), then $\vec{s} = \vec{0}$. If $\vec{s} \neq \vec{0}$, we know an error has occurred. The syndrome information helps us correct the error.

We can understand linear codes in terms of the input and output spaces of the encoding function $\text{Enc}(\vec{x}) = \vec{x}G$. Left multiplication of G by a k-dimensional row vector produces a linear combination of the rows of G. Thus, the set of all possible codewords (called the *code space*) corresponds to the row space of G.

Every vector in the null space of G is orthogonal to every codeword $\vec{c}_i$. We can construct a parity-check matrix H by choosing any basis for the null space for G. We call H the orthogonal complement of G, which means $\mathcal{N}(G) = \mathcal{R}(H)$. Alternately, we can say the space of n-dimensional bitstrings decomposes into orthogonal subspaces of valid and invalid codewords: $\mathbb{F}_2^n = \mathcal{R}(G) \oplus \mathcal{R}(H)$. We know $H\vec{c}^{\mathsf{T}} = \vec{0}$ for all valid codewords $\vec{c}$. Furthermore, the *syndrome* obtained by multiplying an invalid codeword $\vec{c}'$ with the parity check matrix $\vec{s} = H\vec{c}'^{\mathsf{T}}$ can help us characterize the error that occurred, and correct it.

Coding theory

The general idea behind error-correcting codes is to choose the 2^k codewords so they are placed far apart from each other in the space $\{0,1\}^n$. If a code has minimum distance $d \geqslant 2$ between codewords, then this code is robust to one-bit errors. To understand why, imagine a bubble of radius one (in Hamming distance) around each codeword. When a one-bit error occurs, a codeword will be displaced from its position, but it will remain within the bubble of radius one. In other words, if a one-bit error occurs, we can still find the correct codeword by looking for the closest valid codeword. See Figure 7.15 for an illustration of a set of codewords that are $d > 2$ distance apart. Any bitstring that falls within one of the bubbles will be decoded as the codeword at the centre of the bubble. We cannot guarantee this decoding procedure will succeed if more than one errors occur.

Observation 1 An (n,k,d)-code can correct up to $\lfloor \frac{d}{2} \rfloor$ errors.

The notation $\lfloor x \rfloor$ describes the *floor* function, which computes the closest integer value smaller than x. For example, $\lfloor 2 \rfloor = 2$ and $\lfloor \frac{3}{2} \rfloor = \lfloor 1.5 \rfloor = 1$. We can visualize Observation 1 using Figure 7.15 by imagining the radius of each bubble is $\lfloor \frac{d}{2} \rfloor$ instead of 1.

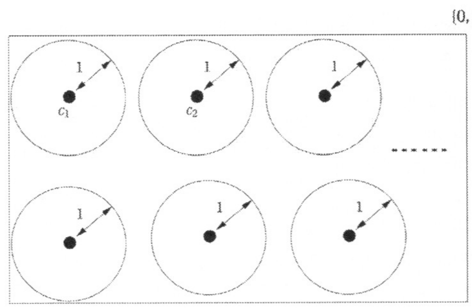

Figure 7.15: The rectangular region represents the space of binary strings of length n. Each codeword c_i is denoted with a black dot. A "bubble" of Hamming distance one around each codeword is shown. Observe that the distance between any two codewords is greater than two ($d > 2$). By Observation 1, we know this code can correct any one-bit error ($\lfloor \frac{d}{2} \rfloor \geqslant 1$).

Repetition code

The simplest possible error-correcting code is the *repetition code*, which protects information by recoding multiple copies of each message bit. For instance, we can construct a $(3, 1, 3)$ code by repeating each message bit three times. The encoding procedure Enc is defined as follows:

$$\mathrm{Enc}(0) = 000 = \vec{c}_0, \qquad \mathrm{Enc}(1) = 111 = \vec{c}_1.$$

Three bit flips are required to change the codeword $\vec{c}_0$ into the codeword $\vec{c}_1$, and vice versa. The Hamming distance between the codewords of this repetition code is $d = 3$.

Encoding a string of messages $x_1x_2x_3 = 010$ results in a string of codewords 000111000. We can apply the "majority vote" decoding strategy using the following decoding function Dec, defined by

$$\mathrm{Dec}(000) = 0,\ \mathrm{Dec}(100) = 0,\ \mathrm{Dec}(010) = 0,\ \mathrm{Dec}(001) = 0,$$
$$\mathrm{Dec}(111) = 1,\ \mathrm{Dec}(011) = 1,\ \mathrm{Dec}(101) = 1,\ \mathrm{Dec}(110) = 1.$$

Observe that any one-bit error is corrected. For example, the message $x = 0$ is encoded as the codeword $\vec{c} = 000$. If an error occurs on the first bit during transmission, the received codeword will be $\vec{c}\,' = 100$, and majority-vote decoding will correctly output $x = 0$. Since $d > 2$ for this repetition code, the code can correct all one-bit errors.

The Hamming code

The $(7,4,3)$ *Hamming code* is a linear code that encodes four-bit messages into seven-bit codewords with minimum Hamming distance of $d = 3$ between any two codewords. The generator matrix for the Hamming code is

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

Note that other possibilities for the matrix G exist. Any permutation of the columns and rows of the matrix will be a generator matrix for a $(7,4,3)$ Hamming code. We have chosen this particular G because of the useful structure in its parity-check matrix H, which we'll discuss shortly.

Encoding

We'll now look at how the generating matrix is used to encode four-bit messages into seven-bit codewords. Recall that all arithmetic operations are performed in the finite field $\mathbb{F}_2$. The message $(0,0,0,1)$ is encoded as the codeword

$$(0,0,0,1)G = (0,0,0,0,1,1,1),$$

similarly $(0,0,1,0)$ is encoded into $(0,0,1,0)G = (0,0,1,1,0,0,1)$. Now consider the message $(0,0,1,1)$, which is a linear combination of the messages $(0,0,1,0)$ and $(0,0,0,1)$. To obtain the codeword for this message, we can multiply it with G as usual to find $(0,0,1,1)G = (0,0,1,1,1,1,0)$. Another approach is to use the linearity of the code and add the codewords for the messages $(0,0,1,0)$ and $(0,0,0,1)$: $(0,0,1,1,0,0,1) + (0,0,0,0,1,1,1) = (0,0,1,1,1,1,0)$.

Decoding with error correction

The minimum distance for this Hamming code is $d = 3$, which means it can correct one-bit errors. In this section, we'll look at some examples of bit-flip errors that can occur, and discuss the decoding procedure we can follow to extract messages—even from a corrupted codeword $\vec{c}\,'$.

The *parity-check matrix* for the $(7,4,3)$ Hamming code is

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

This matrix is the orthogonal complement of the generating matrix G. Every valid codeword $\vec{c}$ is in the row space of G, since $\vec{c} = \vec{x}G$

for some message $\vec{x}$. Since the rows of H are orthogonal to $\mathcal{R}(G)$, the product of H with any valid codeword will be zero: $H\vec{c}^{\mathsf{T}} = 0$.

On the other hand, if the codeword $\vec{c}\,'$ contains an error, then multiplying it with H will produce a nonzero *syndrome* vector $\vec{s}$:

$$H\vec{c}\,'^{\mathsf{T}} = \vec{s} \neq 0.$$

The decoding procedure Dec uses the information in the syndrome vector $\vec{s}$ to correct the error. In general, the decoding function can be a complex procedure that involves $\vec{s}$ and $\vec{c}\,'$. In the case of the Hamming code, the decoding procedure is very simple because the syndrome vector $\vec{s} \in \{0,1\}^3$ contains the binary representation of the location where the bit-flip error occurred. Let's look at an example to illustrate how error correction works.

Example Suppose we send the message $\vec{x} = (0,0,1,1)$ encoded as the codeword $\vec{c} = (0,0,1,1,1,1,0)$. If an error on the last bit occurs in transit, the received codeword will be $\vec{c}\,' = (0,0,1,1,1,1,1)$. Computing the syndrome for $\vec{c}\,'$, we obtain

$$\vec{s} = H\vec{c}\,'^{\mathsf{T}} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

The syndrome vector $\vec{s} = (0,0,1)$ corresponds to the binary string 001, which is the number one. Note we count bits from right to left when interpreting the syndrome of an error: bit one is the rightmost bit of the codeword, bit two is the second to last, and so on. This syndrome 001 tells us the location of the error is on bit one of the codeword, which is the rightmost bit. After correcting the error by flipping the rightmost bit, we obtain the correct codeword $\vec{c} = (0,0,1,1,1,1,0)$, which decodes to the message $\vec{x} = (0,0,1,1)$ that was sent.

Let's check the error-correcting ability of the Hamming code with another single-bit error. If a bit-flip error occurs on bit four (counting from the right), the received codeword will be $\vec{c}\,'' = (0,0,1,0,1,1,0)$. The syndrome for $\vec{c}\,''$ is $H\vec{c}\,''^{\mathsf{T}} = (1,0,0)$, which corresponds to the number four when interpreted in binary. Again, we're able to obtain the position where the error has occurred from the syndrome.

The fact that the syndrome tells us where the error has occurred is not a coincidence, but a consequence of deliberate construction of the matrices G and H of the Hamming code. Let's analyze the possible received codewords $\vec{c}'$, when the transmitted codeword is $\vec{c}'$:

$$\vec{c}' = \begin{cases} \vec{c} & \text{if no error occurs} \\ \vec{c} + \vec{e}_i & \text{if bit-flip error occurs in position } i, \end{cases}$$

where $\vec{e}_i$ is a vector that contains a single one in position i. Indeed a bit flip in the i^{th} position is the same as adding one in that position, since we're working in the finite field $\mathbb{F}_2$.

In the case when no error occurs, the syndrome will be zero $H\vec{c} = \vec{0}$, because H is defined as the orthogonal complement of the code space (the row space of G). In the case when a single error occurs, the syndrome calculation only depends on the error:

$$\vec{s} = H\vec{c}'^{\mathsf{T}} = H(\vec{c} + \vec{e}_i)^{\mathsf{T}} = \cancel{H\vec{c}} + H\vec{e}_i = H\vec{e}_i.$$

If you look carefully at the structure in the parity-check matrix H, you'll see its columns contain the binary encoding of the numbers between seven and one. With this clever construction of the matrix H, we're able to obtain a syndrome that tells us the binary representation of where an error has occurred.

Discussion

Throughout this section, we referred to "the" $(7,4,3)$ *Hamming code*, but in fact there exists much freedom when defining a Hamming code with these dimensions. For example, we're free to perform any permutation of the columns of the generator matrix G and the parity check matrix H, and the resulting code will have the same properties as the $(7,4,3)$ Hamming code discussed in this section.

The term *Hamming code* actually applies to a whole family of linear codes. For any $r > 2$, there exists a $(2^r - 1, 2^r - r - 1, 3)$ Hamming code that has similar structure and properties as the *Hamming* $(7,4,3)$ code. The ability to "read" the location of the error directly from the syndrome is truly a marvellous mathematical construction particular to the Hamming code. Other types of error-correcting codes that infer the error from the syndrome vector $\vec{s}$ may require more complicated procedures.

Note that all Hamming codes have minimum distance $d = 3$, which means they allow us to correct $\lfloor \frac{3}{2} \rfloor = 1$ bit errors. Hamming codes are therefore not appropriate for use with communication channels on which multi-bit errors are likely to occur. There exist other code families, like the *Reed–Muller codes* and *Reed–Solomon codes*, which can be used in noisier scenarios. For example, Reed–Solomon codes are used by NASA for deep-space communications and for error correction on DVDs.

Error-detecting codes

Another approach for dealing with errors is to focus on *detecting* [illegible] errors, rather than trying to correct them. Error-detecting codes, [illegible] the *parity-check code*, are used in scenarios where it is possible [illegible] transmit messages. If the receiver detects a transmission error [illegible] occurred, she can ask the sender to retransmit the corrupted [illegible] sage. The receiver will be like, "Yo, Sender, I got your message [illegible] its *parity* was *odd*, so I know there was an error and I want [illegible] send that message again." Error detection and retransmission is [illegible] internet protocols work (TCP/IP).

The *parity-check code* is a simple example of an error-detecting code. The *parity* of a bitstring describes whether the number of [illegible] in the string is odd or even. The bitstring 0010 has odd parity, [illegible] the bitstring 1100 has even parity. We can compute the parity of [illegible] bitstring by taking the sum of its bits—the sum being performed [illegible] the finite field $\mathbb{F}_2$.

A simple $(k+1, k, 2)$ parity-check code is created by appending a single bit p (the parity bit) to the end of every message to indicate the parity of the message bitstring $x_1x_2\cdots x_k$. We append $p = 1$ if [illegible] message has odd parity, and $p = 0$ if the message has even parity. The resultant message-plus-parity-check bitstring $\vec{c} = x_1x_2\cdots$ [illegible] will always have even parity.

If a single bit-flip error occurs during transmission, the received codeword $\vec{c}\,'$ will have odd parity, which tells us the message [illegible] has been affected by noise. More advanced error-detecting schemes can detect multiple errors, at the cost of appending more parity-check bits at the end of messages.

Links

[The Hamming distance between bitstrings]
`https://en.wikipedia.org/wiki/Hamming_distance`

[More examples of linear codes on Wikipedia]
`https://en.wikipedia.org/wiki/Linear_code`
`https://en.wikipedia.org/wiki/Hamming_code`
`https://en.wikipedia.org/wiki/Reed-Muller_code`

Exercises

E7.15 Find the codeword $\vec{c}$ that corresponds to the message $\vec{x} =$ $(1,0,1,1)$ for the $(7,4,3)$ Hamming code, which has the generator matrix G as given on page 417.

E7.16 Construct the $(5,4,2)$ parity check code's encoding matrix G.