Interactions Between Data Compression and Encryption

James Iwamasa

March 8, 2017

Abstract

Every day, insurmountable amounts of data are shuffled around over an expansive web of users. While the infrastructure to facilitate such movements of information is nothing short of a world wonder, this system inherently poses many problems, the two biggest being that of bandwidth and security. While the fields of data compression and encryption are well developed and highly specialized in their own rights, they both involve the act of altering the raw data somehow, and as a result can come into conflict. This paper will explore these complex interactions between data compression and encryption.

1 Introduction

The direction of this paper can be characterized by a single question: Do we compress our data before we encrypt or the other way around? Basic knowledge of standard encryption and compression techniques will imply that, typically, compression should come first. Data compression often works by noticing patterns and redundancies in our data. On the other hand, encryption attempts to make our data unrecognizable gibberish to anyone who doesn't know the secret code by removing all patterns. Thus, if we want our compression to have the most effect, we should not encrypt it first.

One may also have the intuition that compression might actually assist in encryption, since we're still converting our data into something only a computer can effectively decompress. But there have been studies[Kel02, GHP13] showing that compression can actually expose security flaws. In one method, hackers could use the difference in data length after compression to infer the plaintext of http requests (as in expoits CRIME and BREACH) *without knowledge of the encryption technique.*

This conflict of interests between encryption and compression is the main crux of this paper. Now we will explore how the two can interfere with each other and also help each other in various contexts.

2 Definitions

First, we lay out some basic definitions:

Definition 2.1 *Data compression* is the act of taking raw data and shrinking it to facilitate easier transport and storage.

Definition 2.2 A *compression algorithm* performs the compression, while a *decompression algorithm* returns compressed data to its original, uncompressed form.

Definition 2.3 The *compression rate* of some instance of compression is the amount, usually by percent, the algorithm shrank the data.

Definition 2.4 *Data encryption* (in the field of *cryptograpy*) is the act of taking raw data and transforming it such that only the intended end users can use it.

Definition 2.5 We *encrypt* data into its encrypted form, and the recipient *decrypts* it to get the original data.

Definition 2.6 A *key* is some method or string that is paired (in optimal cases, uniquely) with an encoded set of data or an encoding method. The key is then used to decode the data, the assumption being that one could not decode the data without knowing the key.

3 A Basic Example

We start by analyzing the interactions between compression and encryption with an example:
Let us say we have some data string s which is a string of capital alphabetical letters A-Z. We shall now apply both a compression algorithm and an encryption method onto s and observe what happens. We will use *Huffman encoding*[Huf52] as our method of compression, as it embodies many of the major elements of data compression, and a simple *block cipher* for our encryption for the same reasons.

3.1 Huffman encoding

Huffman encoding involves assigning symbols variable length bit representations based on their relative frequency in the string. The general idea is that more common symbols get shorter representations, and less frequent ones are given longer ones. In the case of our alphabet, we would need at least 5 bits per symbol to

represent them all by a simple enumeration (A = 00001, B = 00010, etc). So Huffman encoding's benefit is twofold: Both by only using the number of bit representations we need, and by using the symbol frequencies to greatly reduce the space used by common symbols.
Let's first consider string s:

ABRACADABRA

First we find the relative frequencies of our symbols, which we put in this table:

Symbol	Frequency
A	0.45
B	0.18
C	0.09
D	0.09
R	0.18

Figure 1: Table of symbol frequencies in "ABRACADABRA"

Next, we must construct our Huffman encoding tree, which is a binary tree that we will use to assign bit representations and also to decode our Huffman encoded string. First, we convert all our symbol/frequency pairs into binary tree nodes with a symbol label and a frequency value and put them in a container A. Then, we will go into a loop, constructing our tree until there is only one element in A:

1. Find the two nodes a and b in A whose frequency values are the lowest, and remove them from A.
2. Create a new node α whose children are a and b and whose value is the sum of their frequency values.
3. Add α to A and go back to step 1.

For s, our tree will look like this, where greek letters are used to represent constructed nodes:

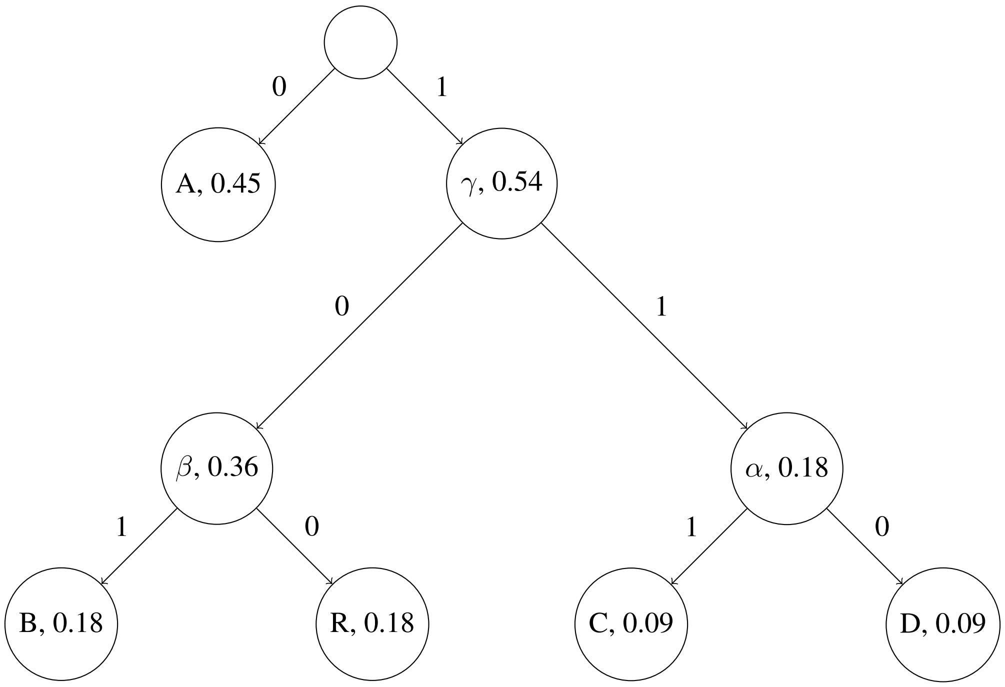

Figure 2: Huffman encoding tree for ”ABRACADABRA”

Symbol	Frequency	Bit code
A	0.45	0
B	0.18	101
C	0.09	111
D	0.09	110
R	0.18	100

Figure 3: Table with bit representations of symbols in ”ABRACADABRA”

Note that each edge in our tree is weighted with either 0 or 1. As shown in Figure 3.1, we get the bit representations of our symbols from the tree by traversing the tree, keeping track of which edge we go down, until we hit a leaf node. So our encoded string $H(s)$ is:

0.101.100.0.111.0.110.0.101.100.0

Which is 23 bits in total, compared to the 55 bits it would have taken if each symbol took a 5 bit int representation per symbol. To decode this, we start at the beginning of $H(s)$ and the root of the tree, reading in symbols and traversing the tree accordingly. When we hit a leaf node, we write the symbol on that node to output, go back to the root of the tree, and resume reading $H(s)$.

Huffman encoding, while not usually used by itself in practice, is a very fundamental and representative compression algorithm for two main reasons: It avoids the problem of unused space, using only the bits we need, and uses the patterns in the input (in this case, the symbol frequencies) to further reduce the amount of information needed.

One may notice that Huffman encoding acts similarly to a encryption algorithm. To decode a Huffman encoded string, one requires the Huffman encoding tree used to encode it, which thus acts like a key. Indeed, one paper[San03] explored a combined compression/encryption technique where the data is first compressed with Huffman encoding, and the resulting tree is then encrypted with a secret key. It is then sufficient to not encrypt the compressed data, as it would be difficult to decode without the tree.

3.2 Block cipher

Now let us study block ciphers, and how we use them to encrypt data. A normal cipher (or ”stream cipher”) involves reading our data string s and replacing the characters one by one to encode our string. ROT13, or the Caesar cipher, is a commonly taught stream cipher where each character in our string is simply shifted by 13 letters. Block ciphers work in a similar manner, only instead of operating on single characters, we operate on blocks of some fixed n characters at once. A block cipher is made of two main elements:

1. A fixed block size n.
2. A *round function*.

To encrypt our string, we break up the string into blocks of size n, and then apply our round function to each block. A round function is usually a collection of simple operations, some of which include:

1. Modulus addition: Adding some key value to each block (like with the ROT13 cipher).
2. Rotations: Shifting elements in the string, wrapping the ends of the block back around.
3. XOR adjacent blocks: After applying some other operations, XOR-ing the current block with, for example, the previous one.

As implied by the ”round” aspect, we may apply any of these multiple times to increase encryption at the cost of runtime efficiency. Decryption is then just running one’s particular sequence of operations in reverse on the encoded string.

For our simple cipher $C()$, we will use a modular addition method similar to ROT13, but using a key instead of a flat, universal constant. We define $C(s)$ with block size n and key k where $|k| = n$ as follows:

1. Assign numeric values 1-26 to each letter (A = 1, B = 2, etc).
2. Divide s into blocks of size n.
3. Add k to each block (that is, add the numeric value of the first character in k to the numeric value of the first character in the block, wrapping around the alphabet, and so on).

Let us perform our cipher on ”ABRACADABRA” with $n = 4$ and $k =$ ”MAGE”:

$$\text{ABRA.CADA.BRA} \rightarrow 1\ 2\ 18\ 1\ .\ 3\ 1\ 4\ 1\ .\ 2\ 18\ 1$$
$$+\ \text{MAGE} \rightarrow 13\ 1\ 7\ 5$$
$$14\ 3\ 25\ 6\ .\ 16\ 2\ 11\ 6\ .\ 15\ 19\ 8 \rightarrow \text{NCYF.PBKF.OSH}$$

Thus our string is encrypted. To decrypt using our simple cipher, we simply break up the encrypted string into blocks again, and subtract off the key from each block.

Note that the letters in the encrypted version of the string are much more randomly distributed than in the original string (which we will quantify in Section 3.4). This is a good property to have for encryption, as obvious patterns may reveal too much information even without knowledge of the key.

3.3 Combining encryption and compression

Now that we have seen what these methods do on their own, let’s see what happens when we apply these two methods together.

3.3.1 Compression then encryption

Let us first try the canonical method of compression before encryption. Using the result of our Huffman encoding, we will compute $C(s)$ with block size $n = 5$ and $k =$ ”10101”. Note that our modular addition here will be just a bitwise XOR operation between k and the block:

$$01011.00011.10110.01011.000 \rightarrow 11110.10110.00011.11110.101$$

This works just as we would expect, as our cipher is not directly affected by the content of the string. Our string is safely encrypted, and still retains the same compression rate from before. Moreover, encrypting after compression is generally faster since we have fewer symbols to encrypt.

3.3.2 Encryption then compression

We now try performing the compression after we have encrypted the data. Using the encrypted string we got earlier, we find our frequencies and construct our Huffman encoding tree:

Symbol	Frequency	Bit code
B	0.09	111
C	0.09	110
F	0.18	000
H	0.09	101
K	0.09	100
N	0.09	0111
O	0.09	0110
P	0.09	0101
S	0.09	0100
Y	0.09	001

Figure 4: Table with bit representations of symbols in ”NCYFPBKFOSH”

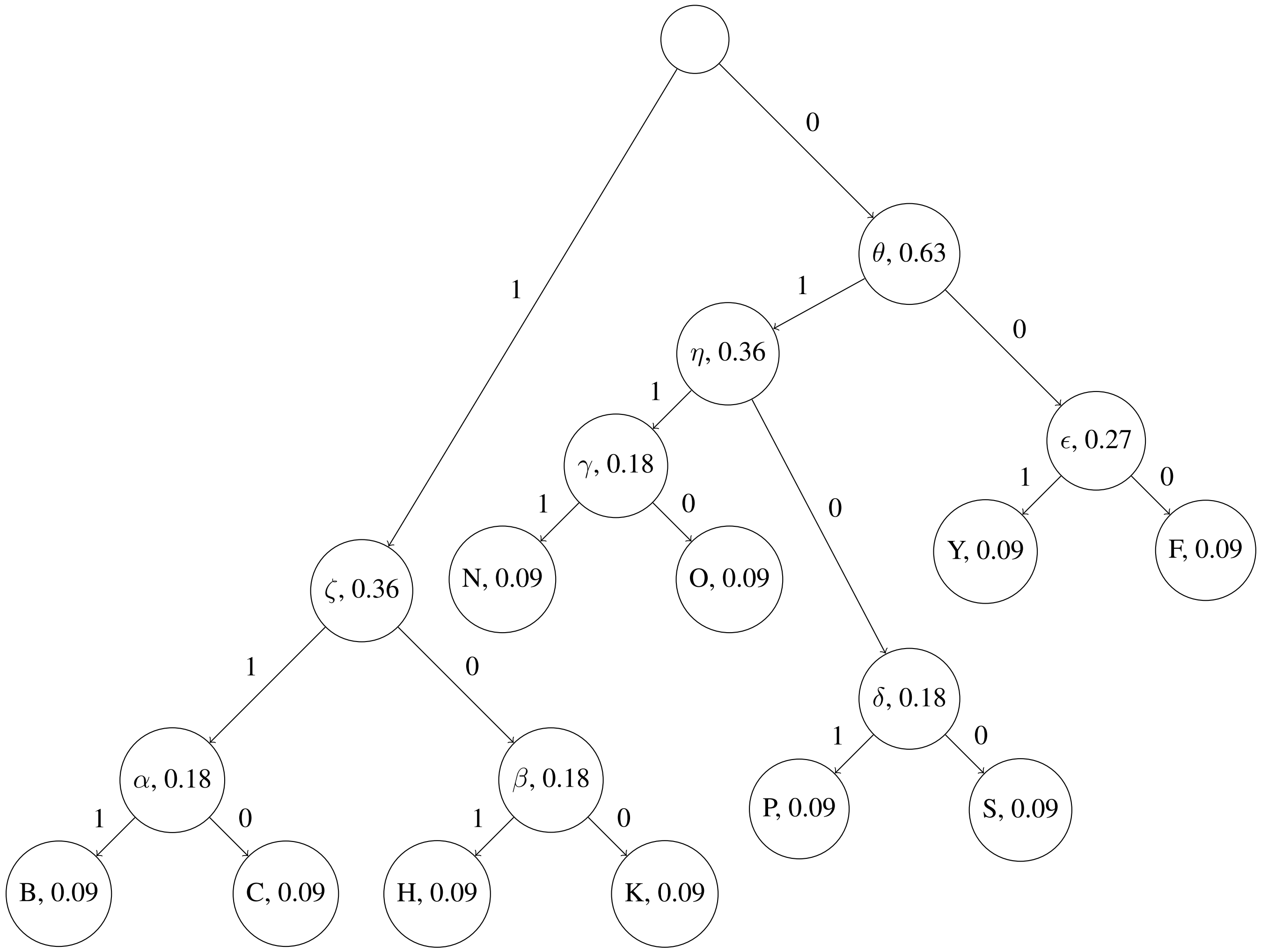

Figure 5: Huffman encoding tree for ”NCYFPBKFOSH”

Thus, we get the compressed data $H(C(s))$:

0111.110.001.000.0101.111.100.000.0110.0100.101

Which is 37 bits long, significantly longer than the string we got as a result of compressing first (which was 23 bits, compared to the normal representation of 55 bits). The reason for this can be easily seen in the Huffman encoding tree we constructed. Huffman encoding assigns bit representations based on relative frequency. When we encrypted with our cipher, the frequencies were scrambled in an evenly distributed way, which, while good for encryption, is not good for compression. So we end up with a much larger tree which, in itself, also adds some overhead, since we need to communicate the tree to the end user so that they can decompress the data.

3.4 Analysis through entropy

We can analyze this effect quantitatively through the use of a concept known as *entropy*. Defined by Claude Shannon in his paper ”A Mathematical Theory of Communication”[Sha48], entropy is the measure of the unpredictability of some event, which implies the amount of information we need to accurately represent the event. If our entropy is 0, then that means we know the outcome of the event without any information (ex. flipping a coin with both sides being heads). But as our entropy gets higher, our events require more information to predict. Mathematically, Shannon defines the entropy H of a set of n events whose probabilities are members of the set $\{p_1, p_2, ..., p_n\}$ as:

$$H = -K \sum_{i=1}^{n} p_i \log p_i$$

Where K is some constant. In the context of our problem, our events are whether a specific symbol comes up with some probability p_i and we use a base of 2 to put our measure of entropy in terms of binary bits.

The conflict between encryption and compression can be summarized thusly: compression typically depends on the original input having low entropy while encryption tends to increase entropy. To show the formor point, let’s look at the case where our input consists of n symbols that all appear equally (with probability = $\frac{1}{n}$. Our entropy for this input is:

$$H = -\sum_{i=1}^{n} \frac{1}{n} \log \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{n} \log n = \log n$$

Note that this result, $\log n$, is the maximum entropy for a particular set of n symbols. What happens when we apply our Huffman encoding scheme to this input? By our definition of the algorithm:

1. First, we run through our set of nodes and pair all of them into sub-trees of 3 nodes. At this point, our set of nodes contains all the parents of the symbols, and they all (in the case where the number of symbols is even) have the same probability. The number of nodes in our set at this point is then $\frac{n}{2}$.

2. We repeat step 1 on the nodes in our set, creating another level of nodes, all with the same probability.

3. We end the algorithm when only one node remains, so we end up with a total of $\log n$ levels in our tree, giving each symbol a bit representation $\log n$ bits long.

The inefficiency of this encoding is apparent if we consider an input alphabet of 128 symbols, that is, the ASCII alphabet. If the frequencies of our symbols were all equal ($p_i = \frac{1}{128}$) as we assumed, our Huffman encoding would give each ASCII symbol a bit representation $\log 128 = 7$ bits long, which is no better than the standard ASCII encoding: exactly what we were trying to avoid by using compression!

This observation is validated by Shannon's Source coding theorem[Sha48] which states that the average length of the codewords (in bits) for some input cannot go lower than the entropy of the original input without some definite loss of information. This applies for all encoding methods, not just Huffman encoding (although it should be mentiond that Huffman encoding does tend to get very close to the theoretical limit in terms of minimum bit length[1]). So the lower our entropy, the smaller our codewords get on average, resulting in a greater level of compression.

A good encryption algorithm, on the other hand, raises entropy. We can see why by looking at the simple Caesar cipher we talked about previously. Again, the Caesar ciper encrypts a string symbol by symbol by shifting each symbol in its alphabet by some constant (the canonical example being by 13 for symbols in the English alphabet). Let's say our string s that we want to encode is a sufficiently large block of natural English text. We will keep our encryption method simple, and say that our key k is the amount by which we shift.

First we note that applying our Caesar cipher does NOT increase the entropy of our string; it merely rearranges the probabilities of the symbols. What this exposes is the fact that, in normal English text, letter frequencies are well defined, allowing a potential hacker to potentially be able to decrypt the string without even knowing the key. This hack can be further improved by the fact that certain letters come up more often after others (”th”, for example, is very common compared to ”qx”) as well as other various syntactic and semantic patterns. By raising entropy, we essentially make the frequencies of each symbol closer to each other, making it difficult to find such patterns.

So how does our simple modulus addition block cipher hold up to this model? Let's say we have a block cipher $C()$ with block size n and key k which is some string of length n. Let's also assume our string s uses symbols from an alphabet of size L, and that our original symbol frequencies $= \{p_1, p_2, ..., p_L\}$. We also add that the choice of our key is totally random (not something silly like ”PASSWORD”).

First, let's look at the ideal case (we will discuss why this is ideal later) where $|s| = n$, in which the

[1]In general, Huffman encoding works better as the individual symbol frequencies reach some inverse of a power of 2 to avoid wasted space from ”fractional bits”. Other methods try to solve this by encoding groups of symbols rather than just individual symbols.

”block frequencies” (the relative symbol frequencies within a block) are equal to the symbol frequencies of the whole input. The probability that some symbol x_i occurs in our encrypted string $C(s)$ is then completely random, since we assumed our key is random. So, our new frquencies are just:

$$p_i' = \frac{1}{L}$$

Plugging this into our forumla for entropy gives us:

$$H' = -\sum_{i=1}^{n} \frac{1}{L} \log \frac{1}{L} = \log L$$

Which is the maximum entropy possible for this particular input. Thus, our Huffman encoding scheme would fail to compress this beyond what standard symbol encodings would give us.

This ideal case acts essentially the same as an encryption method called a ”one-time-pad” cipher. Shannon theorized how one could achieve ”perfect-secrecy” by having a key which uniquely and completely randomly transformed every individual lexical element of the message[Sha49]. If done correctly, the encrypted message would be equivalent to completely random information, and should have a maximum possible entropy for that message. Our ideal modular arithmetic cipher applies a random shift to each symbol in the original string, making it effectively uncrackable without knowledge of the key.

Block ciphers in general (when $|s|$ is some multiple of n), however, are not nearly as strong. Consider a block cipher with $k =$ ”CAT”. Now, consider encrypting this string with our cipher:

DOGDOGDOGDOG→GPAGPAGPAGPA

In this case, we can actually use our previous result to find the new entropy, since each block is identical (the frequencies would be the same if $s =$ ”CAT”). But while the new entropy is the same, a vigilant hacker could realize the repeating pattern, infer the block size, and brute force search through keys of the length of the block size. We can find out how likely it is that this happens fairly easily. Given a string s, block size n (for simplicity, assume that $|s|$ is a multiple of n), and alphabet of size L, the expected number of repeated blocks is:

$$E = \sum_{i=2}^{|s|/n} (i) \binom{|s|/n}{i} \left(\left(\frac{1}{L}\right)^n\right)^i \left(1 - \left(\frac{1}{L}\right)^n\right)^{(|s|/n)-i} = \left(\frac{1}{L}\right)^n \left(\frac{|s|}{n}\right) \left[1 - \left(1 - \left(\frac{1}{L}\right)^n\right)^{(|s|/n)-1}\right]$$

This problem can be generalized to the problem of ”how many times will a group of t symbols come up multiple times in the same positions in different blocks?”, as strings like ”<u>DOG</u>FOOBAR<u>DOG</u>” and

”D$\underline{\text{O}}$GC$\underline{\text{O}}$DR$\underline{\text{O}}$MS$\underline{\text{O}}$W” are also somewhat vulnerable. Our general formula is thus:

$$E = (n+1-t)\sum_{i=2}^{|s|/n}(i)\binom{|s|/n}{i}\left(\left(\frac{1}{L}\right)^t\right)^i\left(1-\left(\frac{1}{L}\right)^t\right)^{(|s|/n)-i}$$

$$= (n+1-t)\left(\frac{1}{L}\right)^t\left(\frac{|s|}{n}\right)\left[1-\left(1-\left(\frac{1}{L}\right)^t\right)^{(|s|/n)-1}\right]$$

As n approaches $|s|$ (approaching our ideal case), or equivalently, as the total number of blocks in our string goes down, we see that the number of repeats goes down (which is intuitive, since the more blocks we have the more chances we have to see repeats). So to reduce the chance of repeats occuring, we would like to increase our block size as much as possible.

But note that repeats and block size don’t have as a direct sway in the entropy of the data. In the case of something like the block cipher, we actually need to consider the entropy of our keys rather than the data, since, as we saw with our first example with ”ABRACADABRA”, our strings do end up fairly random in general. A block cipher is very vulnerable when the block size is known, and finding out the block size can be done by finding repeated patterns (since all blocks are operated on the same way, similar to why our Caesar cipher fails). For a key of length b bits, we only have b bits of entropy at most, and since it’s inefficient to have a key of any signifcant length comparable to the input, simple block ciphers like the one we used are typically too weak to use on their own. However, if we compress our data first, our key becomes larger compared to the size of the data, thus increasing the strength of our algorithm, further pushing our canonical paradigm.

3.5 A quick runtime perspective

As mentioned in Section 3.2, encryption is done by applying a round function of several simple operations on a string in multiple rounds. We can thus increase the strength of our encryption almost arbitrarily just by applying more rounds, but this has a serious runtime cost. Especially if you have a large amount of sensitive data to encrypt, the speed of your encryption can become just as important as its effectiveness. Likewise, the speeds of our compression algorithms are just as much a consideration as the compression rate.

In our example, the speed of our Huffman encoding algorithm was greatly slowed down when applied after the encryption (as can be seen in the difference in sized of the Huffman encoding trees). The encryption in this case was actually faster because of an abstraction we made that allowed us to perform modular addition directly on the symbols themselves rather than their bit representations. However, in a practical scenario we should have stayed consistent, and reconverted the compressed bit string representation into whatever characters they would have represented in, say, ASCII. In this case, and in general, it should be obvious that it's faster to apply an encryption algorithm on a shorter, compressed string rather than an uncompressed one.

4 Non-canonical methodologies

The previous section gave us a simple example showing us why we generally compress before encryption. In our example, not only was the compression much more effective, but overall runtime was improved. However, this can actually expose security flaws.

4.1 CRIME and BREACH: The Reason

CRIME and BREACH[Kel02, GHP13] are two recent security exploits discovered where hackers use the difference in length from compression to infer plaintext from http responses. As a result, even without knowledge of the encryption method, one could find out information about the response.

Here's the basic idea: Let's say there is some secret key k within a sequence of http requests and responses that you want to find out. Most http responses are compressed by the same algorithms, one of the main compression methods being a so called ”dictionary” encoding scheme, where one takes sequences of characters that are the same, and replaces them all with references to a single instance of that string. Say our response contains the string ”What is that hat?”. This form of encoding would convert that to: ”W%1 is t%1 %1?”, where ”%1” is a reference to the string ”hat”.

So how can we use this to find k? Let's say we can isolate the key from the rest of the response. What we then do is inject either short strings of characters or single characters into the responses, and see how long the compressed response is. The information we get is then how much our ”guess” matched up with the actual key. For example, let's say that we make a lucky guess, and inject a near copy of k into the response that has k in it. We then have something like this, where the injected text is in bold:

Original response: my_key=password **my_injection=passwort**

Length = 39

Compressed response: my_key=%1d **my_injection=%1t** (%1=passwor)

Length = 29

Now, we try injecting another string slightly different from our previous injection:

Original response: my_key=password **my_injection=password**

Length = 39

Compressed response: my_key=%1 **my_injection=%1** (%1=password)

Length = 27

Note that we could have cycled through all possible characters for the last symbol in our injection, and we would know when we got the right one just by the difference in size from compression (from 29 to 27). In general, this method allows us to build up character by character hidden secrets in http responses without even knowing the encryption algorithm.

4.2 Compressing encrypted data

While there are specific counter-measures for CRIME and BREACH (not using http compression, special encryption, etc), the issue of information leakage through compression can potentially spread to other means to communications. To solve this problem, we will now look at a method of efficiently compressing data after encryption as described in [JIP$^+$04].

As shown in Section 3.4, compression is limited inversely with the entropy of our data. Since any good encryption algorithm will typically raise the entropy of our data to the theoretical limit, by the aforementioned Shannon's source coding theorem, we cannot create codewords to compress our data by any amount. While the method performed by [JIP$^+$04] does not actually work around this theoretical limit, it cleverly uses the key from the encryption itself as a ”key” to help compress and decompress our data.

4.2.1 Low-Density Parity-Check Codes

The first part of this method uses a concept known as linear codes, and in particular, low-density parity-check (or LDPC) codes[Rob63]. These codes, interestingly enough, are actually more often used to *increase* the size of data for the purpose of making our data recoverable in the presence of noise (for example, in satellite communications). Encoding some message x with LDPC codes requires a few major elements:

1. Integers m and k such that the length of the strings (blocks) we want to encode are of length $m - k$. The choice of m and k is dependant on the data itself, the noisiness of the channel, etc.

2. A *parity-check matrix* H which is $m \times (m-k)$ and sparsley populated. Choice of the elements in this matrix is also dependant on empirical factors, however, a random assignment tends to work well.

First, we must calculate our *generator matrix* G which is achieved by transforming H into the form:

$$\left[\begin{array}{c|c} -P^T & I_{m-k} \end{array}\right]$$

And then converting that into the form:

$$\left[\begin{array}{c|c} I_k & P \end{array}\right]$$

Then, we break x into blocks of size m (labeled b), and our codewords are given as simply bG (labeled $\overline{b}$). Putting all of our codes back together gives us a new message x'.

We then send x' over some noisy channel, where some of the bits become unrecoverable. To decode each noisy block $\overline{b}$ to the corresponding clean block b, we use a method called *belief propagation*, where we iteratively try to figure out each missing bit using our parity-check matrix H. The algorithm works abstractly like this:

1. Treating $\overline{b}$ as a row vector, find a column c with an unknown bit in $\overline{b}$ which satisfies these prerequisites:
   i. There exists at least one row in H whose cth bit is 1.
   ii. In this row, wherever there is a 1 in some other column d, the dth bit of $\overline{b}$ is known.
2. If these are satisfied, then we can figure out the cth bit in $\overline{b}$ by the fact that all of the bits in our original y follow the constraint that the sum of all bits in y specified by a row in H (in the same column as a 1 in that row in H) must be even (equivalent to mod 2). Since we know all the other bits in the constraint, we can easily calculate the remaining unknown bit.
3. Repeat 1 and 2 until all bits are found.
4. Our original b is gotten by taking the first $m-k$ bits of $\overline{b}$. This can be seen in the fact that the first part of our generator matrix G that we used to create the codeword is just the identity matrix.

In more general terms, our parity-check matrix H acts as a set of constraints on the codewords we are allowed (each row being a constraint). Thus, even if some of the bits are lost, we can recover them based on the assumption that the codeword must have passed the constraints.

Of course, one can see that the size of our codewords, if $k > m$ (which is generally the case), is much longer than the original block, which, while intuitively effective for buffering our data against losses, seems completely counter-effective for compression. We require one more method to make this effective.

4.2.2 Distributed Source Coding

The next and last technique we require is related to the *distributed source coding problem*[JIP+04]. The problem involves trying to compress a message/stream E when we have access to another stream F that is correlated to E (that is, some known percentage of bits are the same in E and F). The basic idea is that if we have F, then we can reduce the information needed to identify blocks from E. Or, using entropy, we use the fact that the *conditional entropy* of E is less when we know F, allowing us to compress further.

First we define the *Hamming distance* between two binary strings as a metric for measuring similarity:

Definition 4.1 The *Hamming distance* of two binary strings i and j is the number of bits that they differ in. For example, the strings "1001" and "1010" have a Hamming distance of 2.

We will now describe a simple solution to the distributed source coding problem using cosets which will be adapted (along with LDPC codes) for our solution to compressing encrypted data: Let's say we have a message E which will be broken up into blocks b^E of size n. Let's also assume that the receiver of our message already has the correlated stream F with blocks b^F. We first split up all possible bit strings of length n into "cosets", where each coset will, in our example, contain 2 strings whose Hamming distance is n. For example, if $n = 3$, our cosets would be:

$$\{000, 111\}, \{001, 110\}, \{010, 101\}, \{100, 011\}$$

The union of our cosets would be the set of all possible bit strings of length n. We can then index our cosets, for any n, using $n - 1$ bits (in the $n = 3$ example, we have 4 cosets, so we need 2 bits to index them). To compress, we simple convert each block b^E into the index of the corresponding coset that b^E is in to get $\overline{b^E}$. The receiver then can narrow down the options to two possible strings for each compressed block $\overline{b^E}$ by looking up the coset. However, one can easily see that without any more information, the decoder cannot deduce any further.

This is where F comes in. Let's say that we know that E and F are correlated s.t. every block b^F has a Hamming distance of at most 1 from its corresponding b^E. That is, if some $b^F = 101$, then the options for b^E are $\{101, 100, 111, 001\}$. Our decoding works as follows:

1. Given $\overline{b^E}$, find the coset it refers to.

2. Of the strings in the coset, find the string that is closest (in terms of Hamming distance) to the corresponding b^F.

3. That string is the original b^E.

This works because, since we know that the distance between the two coset elements is n (the largest possible distance), and the distance from b^F to b^E is at most one (which is very small), b^F will also be closer to element in the coset which matches up with b^E. In this sense, like with our Huffman encoding algorithm, we create a ”key” for our compression which provides extra information to be able to decode the compressed string.

However, if the decoder did not have access to F initially, and we had to send it like a key, then our compression fails, since the key would be just as long as the original message, and that would cancel out the compression.

4.3 A Method for Compressing Encrypted Data

We notice that both LDPC codes and our solution to the distributed source coding problem with cosets do not really solve the problem of compression in general, let along compressing encrypted data. Combined, however, they provide us a way of reliably compressing encrypted data by a substantial amount.

The method described in [JIP$^+$04] is as follows:

First, we encrypt our bit string s using a stream cipher where our key k is a bit string of random bits of length $|s|$. Let's also assume that we can communicate this key by simply sending the seed to a random number generator, rather than the whole string. Since each bit is being changed randomly, our entropy will very closely approach 1, which is the maximum entropy for a binary alphabet, insinuating that we cannot compress it at all.

Next, we compress s by running our LDPC encoding method, but in reverse. Instead of finding a generator matrix G and multiplying that by our blocks b of size n to get our encoded strings (which would, of course, just make them longer anyways), we instead encode by multiplying our block directly by the parity-check matrix H. What this gives us amazingly is a compressed bit string which corresponds to the index of a coset with respect to the LDPC code. Our cosets, instead of containing the set of all strings with the same Hamming distance, contain the set of all strings that satisfy the same constraints specified in the rows of H.

At this point, we now have a compressed string where each block was converted into the index of the coset that contains that block (similar to what we had in our distributed source coding problem, but our cosets are much larger and more complex). But how do we decode? In our use of cosets, we required that we had a correlated stream to figure out which element in the coset was closest. We look no further than the key k we used to encrypt. k is indeed correlated to our encrypted string, somewhat obviously by the fact that we used k to encrypt it.

We then use our key k to jointly decompress and decrypt our string (since the key applies to both) using a method similar to belief propagation.

5 Conclusion

Encryption and compression, or rather the respective problems of data security and transportation, are problems that have the same end solution: to change the data itself. However, as we saw in Section 3, the ways in which these two classes of algorithms change data often come into conflict, as compression attempts to find patterns and simplify them, while encryption attempts to remove all patterns. And while this suggests that we should compress before we encrypt, we also saw how compression can hurt encryption through exploits like BREACH. But by using compression methods that are conscious of the encryption (or vice-versa)[JIP+04, San03], we can greatly improve compression rates, secrecy, and efficiency. While coming up with such algorithms takes a lot of enginuity, as data becomes larger and more prominent in our lives, more and more consideration will have to be made on the complex interactions between encryption and compression.

References

[GHP13] Y. Gluck, N. Harris, and A. Prado. Breach: Reviving the crime attack. Technical report, July 2013. [accessed February 2017].

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. In *The Proceedings of the I.R.E.* IEEE, September 1952.

[JIP$^+$04] M. Johnson, P. Ishwar, V. Prabhakaran, D. Schonberg, and K. Ramchandran. On compressing encrypted data. *IEEE Transactions on Signal Processing*, 53(10), October 2004.

[Kel02] J. Kelsey. Compression and information leakage of plaintext. In *Revised Papers from the 9th International Workshop on Fast Software Encryption*, pages 263–276, London, UK, February 2002. Springer-Verlag.

[Rob63] G. Gallager Robert. *Low Density Parity Check Codes*. M.I.T. Press, 1963.

[San03] N. Sangwan. Text encryption with huffman compression. *International Journal of Computer Applications*, 54(10), October 2003.

[Sha48] C. E. Shannon. Mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, July 1948.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, October 1949.