Interactions Between Data Compression and Encryption

James Iwamasa

February 23, 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. 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).

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, peacefully coexist, and perhaps even help each other in protecting and packaging our data.

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 normal means (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 C. Then, we will go into a loop, constructing our tree until there is only one element in C:

1. Find the two nodes a and b in C whose frequency values are the lowest, and remove them from C.
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 C and repeat.

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

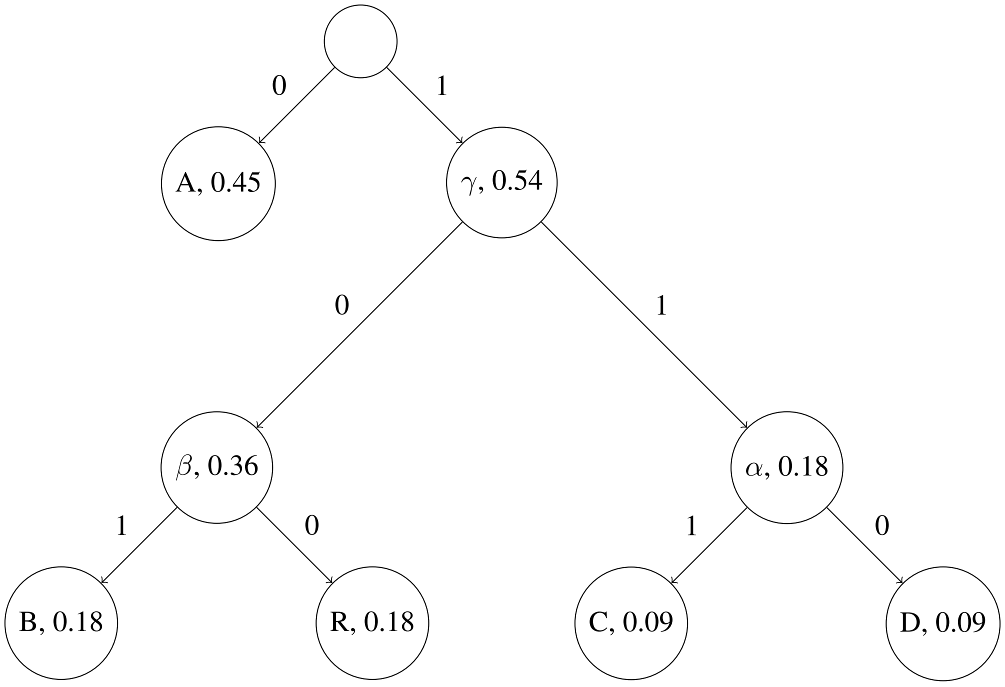

Figure 2: Huffman encoding tree for ”ABRACADABRA”

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. Our 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 $n = 4$ 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 block-size = 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. This is a good property to have for encryption, as obvious patterns may reveal too much information.

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.

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”

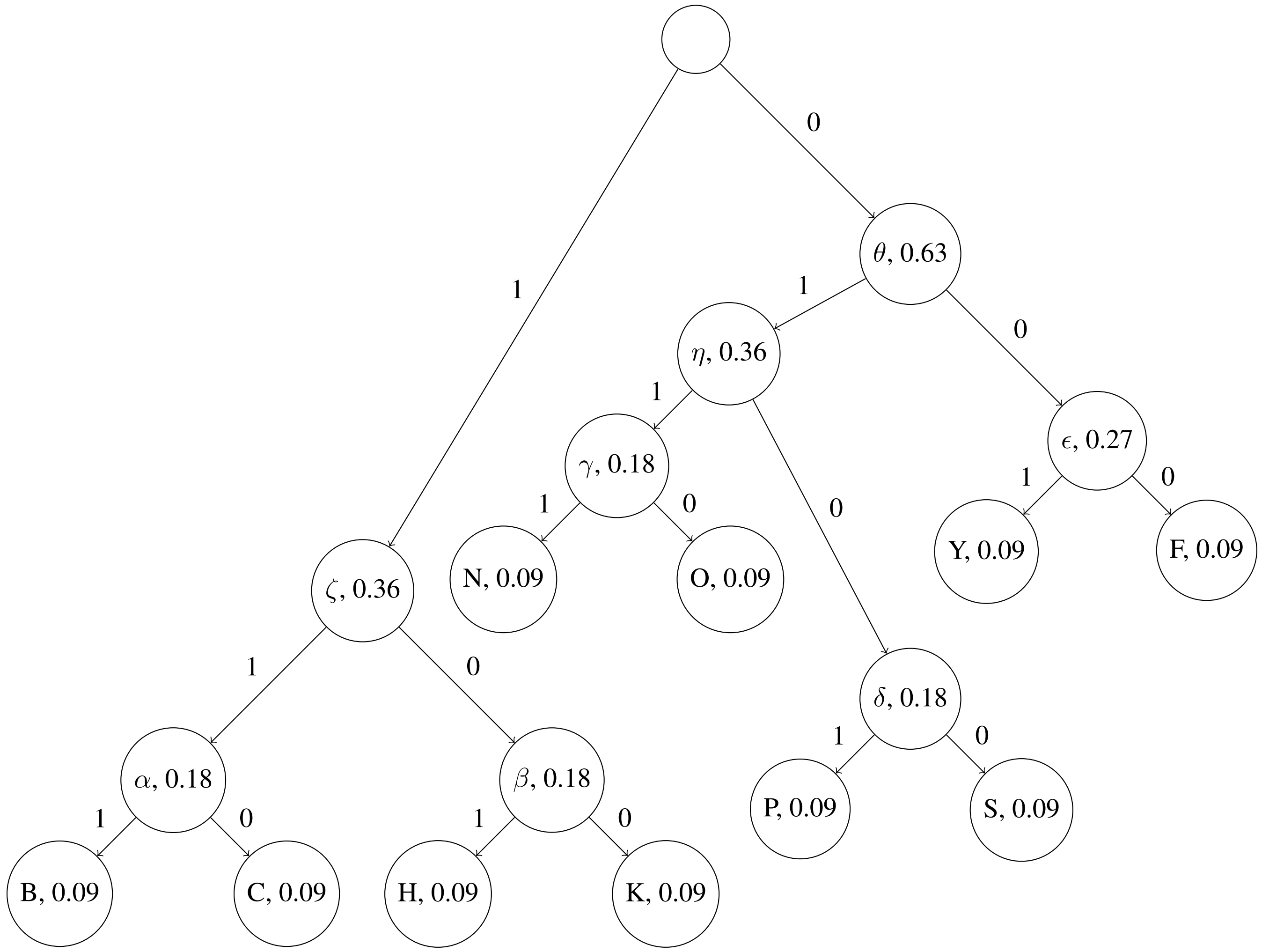

Figure 5: Huffman encoding tree for "NCYFPBKFOSH"

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.

In our simple example, we have shown that the canonical compression before encryption approach is better than the alternative for the algorithms we chose, but there are many different kinds of compression and encryption algorithms. We will now explore these other interactions, and see how the canon holds.

4 A 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 to a sufficient degree 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, while the speed of our simple cipher did not really depend on the data itself, the speed of our compression algorithm was greatly slowed down when applied after the encryption (as can be seen in the difference in sized of the Huffman encoding trees). Certainly, the compression still compressed the data to a certain degree, but at a severe runtime cost.

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.

[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.

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