Rosie Bartlett

FILE COMPRESSION USING SUCCESSIVE SINGLE KEY BIT-SHIFTED XOR FUNCTIONS

Abstract *Lossless compression is exceptionally useful for data storage on small and large scales. In this paper, I would like to propose a new method for lossless data compression using single key bit shifting and successive XOR functions. This paper is primarily a proof of concept paper to demonstrate the capabilities of this method, which would undoubtedly improve with further research.*

Keywords compression, bit shifting, single key, XOR

1. Introduction

Throughout this paper I will be exploring a new compression method, named SBX(successive bit-shifted XOR) with the file extension `.sqzr`. SBX takes in a batch of binary strings, and returns a master binary string, set of keys, and set of bit shift lists. SBX is based upon the concept that the master binary string is a general overview of all given binary strings. This overview is then "corrected" by long keys to make the master binary string into the relative binary string for that long key. This long key can be represented as a shorter key, k_n and list of bit shifts, s_n, which is how SBX achieves compression.

One key part of SBX is bit shift lists. A bit shift list (BSL) is simply a list of integer values that is used along with a key to represent a long key. Each item in the list is referred to using the notation $s_{n,i}$ for the i^{th} value in the bit shift list associated with file n. Please note that the term file and binary string will be used interchangeably throughout since they are considered the same within this scope.

Something to sum up the results section and show off how good it is.

Symbol	Description	Type
s_n	Bit shift list for file n	List of integers
k_n	The associated key for file n	Bit string
$D(k_n, s_n)$	Decoding function. See equation (1)	Function
Ξ	Bitwise XOR shorthand. See equation (2)	Operation
$<<$	Bit shift left	Operation
I_n	Intermediate key, or long key. See equation (3)	Bit string
F_n	File n, where n$\neq 0$	Bit string
F_{0C}	Calculated F_0 for a given key and BSL pairing. This may be an incorrect F_0	Bit string
F_0	Compressed file	Bit string
$L(s)$	Number of items in all bit shift strings	Integer

Table 1

Notation used throughout this paper, in order of occurrence

2. Method

SBX compresses files by using a single master file and compressed keys. The compressed keys is how SBX reduces the volume of data stored. An example of this key compression is shown in section 2.1.1. Key compression uses shorter keys and BSLs to represent longer keys, shown in equation (1).

Since this method involves multiple XOR functions, a shorthand function for recurrent XOR functions has been defined in equation (2).

$$D(k_n, s_n) = \mathop{\Xi}_{i=1} (k_n << s_{n,i}) \tag{1}$$

$$\mathop{\Xi}_{i=1}^{m} (x_i) = x_1 \oplus x_2 \oplus \cdots \oplus x_m \tag{2}$$

$$I_n = F_n \oplus F_{0C} \tag{3}$$

2.1. Compression

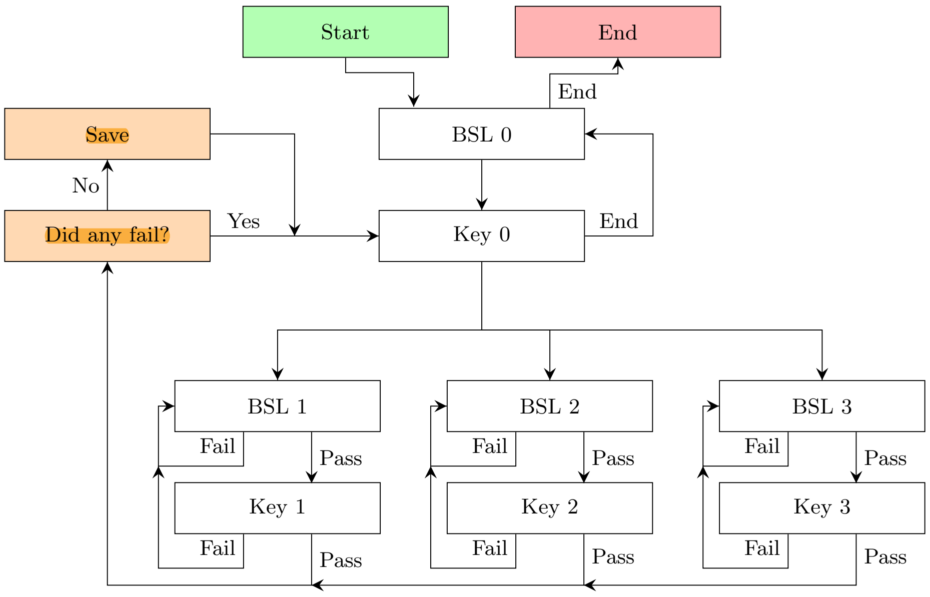


Figure 1. Flowchart of one compression cycle including the beginning compression cycle

Compression using SBX is a semi-iterative parallel process. It begins by iterating over all possible values for the first BSL, s_0, and for each iteration iterating over all possible values of the first key, k_0. Fir each of these iterations, F_{0C}. This is distinguished from F_0 to avoid confusion, since F_{0C} is not F_0 for the majority of the iterations. F_{0C} is then passed to each of the BSL and key pairs remaining, and the BSL in these pairs begins iterating over all of the possible values it could take, each time running the following checks in the order given:

1. The length of the current BSL's current value is calculated and added to the cumulative length counter held by the master BSL and key pair. This counter holds the sum of the length of all the BSL's for BSL and key pairs that managed to return a successful compression. This value is then added to the number of files given, n, minus the index of the BSL plus 1. If this value is less than the previous lowest sum of all the lengths of the BSL's, then this check is passed.

2. Next the first value in the current BSL is compared with the index of the first non-zero bit in I_n. This check is passed if the first value in the current BSL plus the length of the key, is less than or equal to the index of the first non-zero bit in I_n . This check is then performed with the last value in the current BSL and the index of the last non-zero bit in I_n, but instead of being less than or equal, the last value in the BSL must be greater than or equal to the index of the last non-zero bit in I_n.

If either of these checks are not passed for any of the potential values of the BSL, a new F_{0C} is requested, and the process begins from the first key. If one of the checks are failed before all possible values have been iterated over, a new BSL is assigned and the checks begin again.

Once these checks have been passed, the current BSL is considered to be a potentially correct BSL, and the process is then passed to the key related to that BSL. The key then performs checks based upon the length of the BSL. If the BSL has a length of 1, then the key is simply assigned to $I_n >> s_{n,0}$ since it has already been established previously that the current BSL is a viable one. If the BSL does not have a length of 1, the following is performed:

1. The key is initialised with a value of 0.
2. The key is then calculated bit by bit by XOR-ing I_n with $D(k_n, s_n)$. Even though k_n is incomplete, this works bitwise, allowing the successive bit to be calculated next, and thus calculating the entire key. In this context, $D(k_n, s_n)$ is made more efficient by checking the index of the current value in the BSL s_n, and ensuring that it is lower than current index of the bit being calculated thus removing unnecessary calculation.
3. This method of finding a key does not work perfectly, and thus the next step is a check to ensure that the calculated key has been calculated correctly.

If the above is successfully completed and the correct key is calculated, then the key returns true along with the length of the BSL used for this compression. In practice this is not entirely true, since the key actually returns true to it's related BSL which then returns the required data to the master key, but the process still results in the same result. If the key does not manage to calculate a key for the BSL, a new F_{0C} is requested and the key return false. If true is returned to the master key, then the next BSL and key pair is started calculating the required BSL and key values for their respective file. This method is particularly interesting since the iteration cycles can be run independently for the BSL and key pairings after the master pair, and thus SBX compression can make use of cluster computers to make the process more time efficient. Figure 1 shows the whole compression cycle in a flowchart-like style for 4 files. Unlike a typical flowchart, the orange boxes show both decisions and functions, while the white boxes represent either BSLs iterating over all possible values, or keys. The first key also represents an iterative process. This has been done to compress the process into a smaller space without the need for a large number of decision nodes to represent iteration.

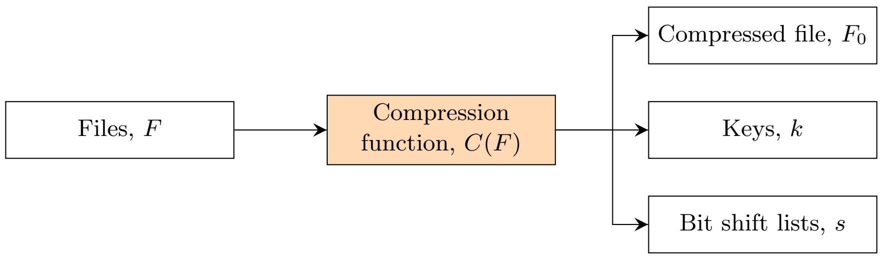
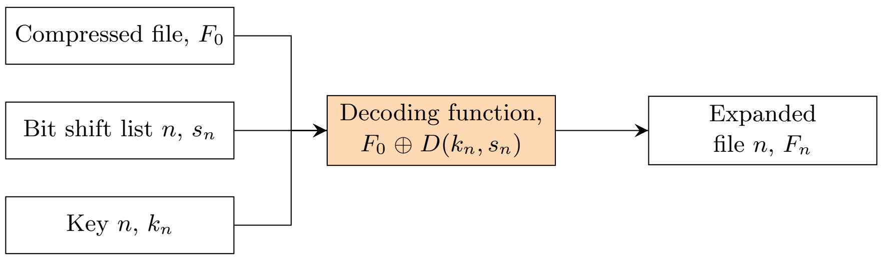


Figure 2. Visual representation of compression

2.1.1. Example long key compression

Given F_0=`10110101` and F_n=`11001000`, therefore the long key must be `01111101`. However, this long key can be compressed, and represented using the key `11001` and the BSL `[0, 2]`. To expand the key and BSL pair into the long, or intermediate, key, which is XOR-ed with the master file to produce the file related with this specific BSL and key pair, the BSL is used one item at a time in the following manner once the long key has been initialised to 0. First the key is bit shifted left by the current value in the BSL. This is then XOR-ed with the long key and replaces the previously stored long key value. Using the example previously given, this would result in `11001` being first bit shifted by 0 to the left resulting in `11001`. This is then XOR-ed with the initialised long key value of 0, resulting in the long key now having a value of `11001`. The next value in the BSL is 2, so the key is bit shifted left by 2 resulting in `1100100`. This is then XOR-ed with the long key resulting in `01111101`, which is the desired long key for this F_0 and F_n.

2.2. Expansion



Compressed file, F_0
Bit shift list n, s_n
Key n, k_n
Decoding function, $F_0 \oplus D(k_n, s_n)$
Expanded file n, F_n

Figure 3. Visual representation of expansion

The expansion method of SBX is incredible simple, requiring only a few bit shifts left and XOR functions. To expand a file, the master file F_0 is XOR-ed with $D(k_n, s_n)$, resulting in F_n, showin in equation (4).

$$F_n = F_0 \oplus D(k_n, s_n) \tag{4}$$

3. Evaluation

To evaluate the effectiveness of this method, I will be using two methods; one including keys, and one without. This is meant to represent one of the end goals of this method, having memorable and human-usable keys, which would not have to be stored. From here on, these will be referred to as EM1, with keys, and EM2, without keys (for evaluation method 1 and 2 respectively). Both methods will be use five measures to evaluate the efficiency of this compression method. These are as follows:

- S: A ratio of the number of batches compressed using SBX compared to the total number of batches passed in
- ΔB_s: The average change in bits stored for the successful batch compressions
- ΔB: The average change in bits stored for all batches regardless of whether or not the batch was considered to be compressed successfully
- R_s: A ratio of the average size of a compressed file compared to the original files for all successful compressions
- R: A ratio of the average size of a compressed file compared to the original files for all compressions regardless of success

Please note that for files with a length of approximately 16 bits or more, ΔB_s and ΔB, and R_s and R are the same and so the value will span two columns in the results tables. Also, ΔB_s and ΔB are signed values; a negative value indicates a reduction in bits, and a positive value indicates an increase in bits.

To evaluate S for EM1 and EM2, equation (5a) and equation (5b) are used respectively, where n is the number of files in the batch, and $2l$ is the length of the largest input files in bits.

For all of the input files, F, the longest file will determine the size of all the output files. In testing this method, all files were the same size. Given that the longest input file has a size of $2l$ bits, the master file will also have a size of $2l$ bits, each key will have a size of $l+1$ bits, and each value in each bit shift list will have a size of $\log_2 l$ bits.

$$L(s) < \frac{nl - 2l - n}{\log_2 l} \tag{5a}$$

$$L(s) < \frac{2l(n-1)}{\log_2 l} \tag{5b}$$

I will also be looking at the values of $L(s)$, minimum mean and maximum. This is to help evaluate the best value of n for SBX to be most efficient.

4. Results

For each of the results tables, 1000 iterations were used. All decimals are given to five decimal places if necessary.

l (Bits)	EM1					EM2				
	S	ΔB_s	ΔB	R_s	R	S	ΔB_s	ΔB	R_s	R
4	0	N/A	5	N/A	1.20833	1	-7		0.58333	
8	0	N/A	4	N/A	1.08333	1	-20		0.52083	
16	?	?	?	?	?	?	?	?	?	?
32	?	?	?	?	?	?	?	?	?	?

Table 2

Compression results for 4, 8, and 16 bit strings for EM1 and EM2 with a batch size of 3.

l (Bits)	EM1					EM2				
	S	ΔB_s	ΔB	R_s	R	S	ΔB_s	ΔB	R_s	R
4	0	N/A	4.468	N/A	1.13963	1	-11.532		0.51462	
8	?	?	?	?	?	?	?	?	?	?
16	?	?	?	?	?	?	?	?	?	?
32	?	?	?	?	?	?	?	?	?	?

Table 3

Compression results for 4, 8, and 16 bit strings for EM1 and EM2 with a batch size of 4.

l (Bits)	EM1					EM2				
	S	ΔB_s	ΔB	R_s	R	S	ΔB_s	ΔB	R_s	R
4	0	N/A	4.584	N/A	1.1146	0.999	-15.43243	-15.416	0.48919	0.4896
8	?	?	?	?	?	?	?	?	?	?
16	?	?	?	?	?	?	?	?	?	?
32	?	?	?	?	?	?	?	?	?	?

Table 4

Compression results for 4, 8, and 16 bit strings for EM1 and EM2 with a batch size of 5.

Acknowledgements

The research presented in this paper was aided in part by the MPI library for python, [2]. I would also like to thank Rosetta Code for their code published on finding the first non-zero bit in a binary string [1].

References

[1] Rosetta Code. *Find first and last set bit of a long integer*. Jan. 2021. URL: `https://rosettacode.org/wiki/Find_first_and_last_set_bit_of_a_long_integer`.

[2] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. "MPI for Python". In: *Journal of Parallel and Distributed Computing* 65.9 (2005), pp. 1108–1115. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2005.03.010`. URL: `https://www.sciencedirect.com/science/article/pii/S0743731505000560`.

l (Bits)	EM1					EM2				
	S	ΔB_s	ΔB	R_s	R	S	ΔB_s	ΔB	R_s	R
4	0	N/A	5.692	N/A	1.11858	0.99	-18.51313	-18.308	0.48931	0.49358
8	?	?	?	?	?	?	?	?	?	?
16	?	?	?	?	?	?	?	?	?	?
32	?	?	?	?	?	?	?	?	?	?

Table 5

Compression results for 4, 8, and 16 bit strings for EM1 and EM2 with a batch size of 6.

n	l (Bits)	$L(s)$ data		
		Min.	Mean	Max.
3	4	3	3.233	4
	8	?	?	?
	16	?	?	?
	32	?	?	?
4	4	4	4.724	7
	8	?	?	?
	16	?	?	?
	32	?	?	?
5	4	5	6.319	9
	8	?	?	?
	16	?	?	?
	32	?	?	?
6	4	6	8.114	12
	8	?	?	?
	16	?	?	?
	32	?	?	?

Table 6

Caption