Turning Compression Schemes into Crypto-Systems

Kfir Cohen[1], Yonatan Feigel[1], Shmuel T. Klein[1], and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
{kfirco12,feyon9}@gmail.com, tomi@cs.biu.ac.il
[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@g.ariel.ac.il

Abstract. Some techniques are presented turning several of the classical compression schemes into crypto-systems, not only reducing their space, but also securing their content by means of encryption based on the knowledge of a secret key. Only a single bit of the key is consumed at every encoding step, and the cumulative impact of applying these steps is shown to produce almost random output. Only the techniques are presented; security issues are deferred to future work.

1 Background

The ever increasing number of cyber attacks necessitates the protection of all transmitted personal data and, in particular, delicate information. The challenge is to protect the data without hurting the network's throughput rate. A *Compression Cryptosystem*, which performs compression and encryption simultaneously, is meant to overcome these challenges. This combined system is accomplished by either inserting compression tools into encryption algorithms (see [22] for example), or by embedding cryptography into the compression systems (see [8] and [5]).

Encrypted data usually cannot be distinguished from a randomly generated file, as empirically shown by Sharma and Bollavarapu [15] and by Carpentieri [2]. Therefore, when both compression and encryption are desired, compression cannot be applied after encryption, only before or in parallel. Most existing solutions perform compression and encryption sequentially. Contrarily, in this paper we propose a *single* process that produces secure outputs against unauthorized eavesdroppers and also reduces the memory usage. Unlike the compression cryptosystem of [8], that used the secret key to control the coding of the *model*, the current research follows the approach of [5] and controls the code generation itself.

A *One Time Pad* (OTP) is a process for encrypting any message, sequentially, by XORing it with a one-time secret key of length at least the size of the message. Singh et al. [16] apply OTP on a file that has been compressed by arithmetic coding. Empirical experiments show that the processing times of simultaneous compression and encryption are shorter than applying them sequentially. Raju [12] has examined this fact for OTP with arithmetic coding, while Sangwan [13] tested it for a particular algorithm that combines reversed static Huffman codewords with different secret keys. Setyaningsih and Wardoyo [14] provide a thorough survey concerning the combination of cryptography and lossless and lossy compression methods, and they mention that most of such research concentrates on image security rather than on compression efficiency.

Our previous work concentrated on Huffman and arithmetic coding and performed a sequence of small perturbations steps, which cumulatively had the impact of scrambling the output enough to turn the decoding into an (almost) impossible task without

Kfir Cohen, Yonatan Feigel, Shmuel T. Klein, Dana Shapira: Turning Compression Schemes into Crypto-Systems, pp. 104–113.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6 © Czech Technical University in Prague, Czech Republic

the full knowledge of the secret key $\mathcal{K}$. Yet, the number of bits of $\mathcal{K}$ consumed by each perturbation step is of the order of $\log n$, where n is the size of the encoded alphabet. Since we do not limit this alphabet to consist only of single characters, n might in fact be quite large. In many applications, especially when processing large Information Retrieval corpora, the term *alphabet* should be understood in a broader sense, and it may consist of the different words in the textual database, or of other variable length strings, see, e.g., [10] or [1].

We therefore turn in the current work to a different paradigm, in which we constrain the process to use at most a single bit of the secret key at each step, actually, a binary decision between two plausible alternatives. This reduction in the number of required secret bits may come at the price of a possible deterioration of the compression performance. However, our experiments indicate only a negligible loss in compression efficiency, as shown below.

The idea of shuffling elements of the compression model, according to a secret key, has already been explored is several studies. For example, Wang [19] shuffles such elements of several well known compression algorithms in a preprocessing stage. In particular, the initial *dictionary* is shuffled in LZW compression [20], the *order* of the alphabet symbols in the working interval is shuffled in case of arithmetic coding [21], and the Huffman *tree* gets scrambled when Huffman coding is used. A similar approach for Huffman coding, that shuffles the Huffman tree in a preprocessing stage, is performed in [18]. Zebari [23] converts a given message into a corresponding DNA file based on a secret function, followed by compressing the converted message and then encrypting it by an additional shuffling procedure. Kelley and Tamassia [6] suggest a compression cryptosystem based on LZW that periodically alters the dictionary. In our work, following the approach of [5], the shuffles are done iteratively, and not only in a preprocessing or postprocessing stage.

The resulting size of the ciphertext might sometimes become a burden for data protection. BREACH is an attack based on the size of the output ciphertext due to Gluck et al. [4]. Their idea is to gradually try to retrieve encrypted secrets from a HTTPS channel.

For our experiments, we used the King James version of the Bible, of size about 4.25MB, taken from the Gutenberg project[1], and the 50MB variants of the files *english*, *XML* and *DNA* of the Pizza & Chili corpus[2]. The fact that our results were similar on all the tests shows that they are not dependent of the nature of the input files.

In the next section, we explore our new ideas in detail for Huffman and arithmetic coding, as well as for the Ziv-Lempel dictionary based methods LZW and LZ77, and for compression after having applied the Burrows-Wheeler Transform. Section 3 concludes and suggests future work.

2 Adding binary decisions to the compression process

We assume that a secret key $\mathcal{K}$ of sufficient length has been exchanged between sender and receiver prior to the transmission of any encoded message $\mathcal{E}_{\mathcal{K}}(M)$ called the *ciphertext*, and that this key is not known to a potential opponent, whose aim it is to reveal the content of the original message $\mathcal{D}_{\mathcal{K}}(\mathcal{E}_{\mathcal{K}}(M)) = M$, known as the

[1] https://www.gutenberg.org/cache/epub/10/pg10.txt
[2] http://pizzachili.dcc.uchile.cl/

cleartext. To facilitate the description of our methods, we may assume that the size of the key is at least as long as the message itself, though this would then be equivalent to an ideal OTP. In practice, a secret key of any length can be produced by choosing a random seed for some random number generator, of length, say, 1000 bits, so that it cannot be guessed.

The common idea for the following methods is to let the compression algorithm depend on some binary choice, which on the one hand, should not at all, or only slightly, impact on the compression efficiency, but on the other hand, produce a completely different ciphertext. The methods differ in the details on where in the process such a choice may be applied. The challenge is in the design of the algorithm, the correctness and efficacy of which should be oblivious to the actual value of the secret key $\mathcal{K}$.

2.1 Huffman coding

Huffman codes are well known to yield optimal compression once it has been decided which elements to encode. The set of these elements is referred to as the *alphabet*, even if they are not just single characters. The additional constraint for optimality is that all the codewords consist of an integral number of bits, which is not obvious, as it may be overcome by arithmetic coding. There are, however, many different Huffman codes, or, equivalently, Huffman trees, for any single probability distribution. We consider *Mirror* and *Swap* transformations, already mentioned in [5].

Mirror: Given is a Huffman tree T and one of its internal nodes v, or, equivalently, a Huffman code C_T and a proper prefix α_v of some of its codewords. A mirror transformation $M(T)$ of T replaces the subtree T_v rooted at v by its mirror image, in which the left and right branches emanating from every internal node of the subtree T_v are interchanged. Equivalently, the codewords corresponding to the leaves of T_v have their suffixes following the prefix α_v complemented, that is, if $\alpha_v\beta \in C_T$, then $\alpha_v\overline{\beta} \in C_{M(T)}$, for all strings β. Since the lengths of the codewords did not change, their average is still optimal.

Swap: Using the same notation, a swap transformation $S(T)$ of T replaces the subtree T_v rooted at v by a subtree whose own left and right subtrees have changed sides, but without altering their shape. This is like the mirror transformation, but only for one level, without continuing recursively to the sub-subtrees. Equivalently, only the bit following the prefix α_v is complemented, for all the codewords corresponding to the leaves of T_v. That is, if $\alpha_v\gamma\beta \in C_T$, then $\alpha_v\overline{\gamma}\beta \in C_{S(T)}$, for $\gamma \in \{0, 1\}$ and all strings β. Here again, the transformation does not change the lengths of the codewords.

Table 1 shows an illustrative example. The original code is given in the left column, and the chosen internal node v is the right child of the root of the tree, corresponding to the prefix $\alpha_v = 1$. The elements of the alphabet corresponding to the leaves of T_v are shown on shaded background, red or blue for the left or right subtrees of T_v, respectively.

The middle column shows the code after the mirror transformation: all the suffixes, shown in red or blue, of the codewords starting with 1 are complemented, and are then rearranged to keep the lexicographic order. As a result, the left to right order of the alphabet elements corresponding to leaves of the subtree T_v is reversed, as indicated by matching color shades.

Original		Mirror		Swap	
a	0 0	a	0 0	a	0 0
b	0 1 0 0	b	0 1 0 0	b	0 1 0 0
c	0 1 0 1 0	c	0 1 0 1 0	c	0 1 0 1 0
d	0 1 0 1 1	d	0 1 0 1 1	d	0 1 0 1 1
e	0 1 1	e	0 1 1	e	0 1 1
f	1 0 0	m	1 0 0 0	k	1 0 0
g	1 0 1 0 0	l	1 0 0 1	l	1 0 1 0
h	1 0 1 0 1	k	1 0 1	m	1 0 1 1
i	1 0 1 1 0	j	1 1 0 0 0	f	1 1 0
j	1 0 1 1 1	i	1 1 0 0 1	g	1 1 1 0 0
k	1 1 0	h	1 1 0 1 0	h	1 1 1 0 1
l	1 1 1 0	g	1 1 0 1 1	i	1 1 1 1 0
m	1 1 1 1	f	1 1 1	j	1 1 1 1 1

Table 1: *Mirror and swap transformations in a Huffman code.*

The right column corresponds to a swap. Here, only the single bit after the prefix $\alpha_v = 1$ is complemented, which moves the codewords of the left and right subtrees of T_v as solid blocks, without changing their internal order.

While in previous work, $\log n$ bits of the secret key were used at each step to point to one of the internal nodes v of the tree T at which the mirror or swap had to be applied, we now suggest to use just a single bit to decide whether the transformation will be of type mirror or swap. This supposes that the internal nodes of the tree are scanned in some predetermined fixed order, known to both encoder and decoder, and possibly also to a potential eavesdropper. The order could be generated randomly, to avoid anomalies in trees with particular layouts, but it is not assumed to be a part of the secret known only to the communicating parties.

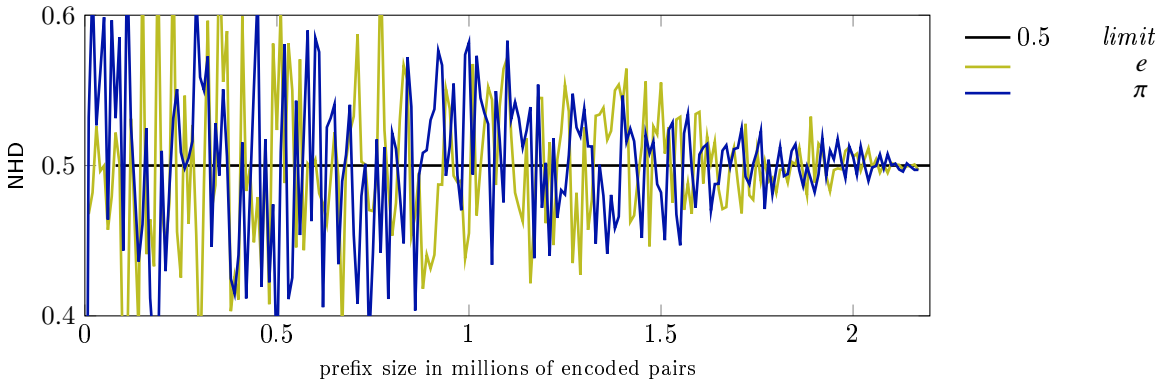

Figure 1. NHD for two runs on the same text with different keys for Huffman coding.

Given two binary files A and B, we define their *Normalized Hamming distance* as

$$\mathsf{NHD}(A,B) = \frac{1}{n}\sum_{i=1}^{n} A_i \text{ XOR } B_i,$$

where X_i is the ith bit of the file X, n is the size of the larger of the two files A and B, and X_i is defined as 0 for indices larger than the size of X. We use the NHD as a measure of the similarity between files, a perfect match corresponding to NHD = 1, while for completely unrelated files, the NHD should tend to $\frac{1}{2}$. Figure 1 shows the effect on the King James Bible of applying the suggested transformations with random keys generated by different seeds. In our example, we have chosen as seeds the first

few bits of the expansion of e or π. The figure plots the NHD between a file obtained by applying mirror and swap transformations and a file using the original Huffman code for all its codewords, as a function of the index within the files. We see that the values fluctuate symmetrically and ultimately zoom in on the expected limit $\frac{1}{2}$. Moreover, the two files using the transformations with different seeds are themselves completely unrelated.

2.2 Arithmetic coding

The output of an arithmetic coder is a subinterval within $[0, 1)$, or just a single real number in it. One starts with $[0, 1)$ and at each step, the current interval is narrowed according to the currently processed character. More precisely, the unit interval is partitioned into regions corresponding to the elements of the given alphabet, the sizes of the regions being proportional to the probabilities of the characters. Arithmetic coding is known to reach entropy, regardless of the initial assignment of the regions to the characters.

In previous work, the suggested transformations were swaps between adjacent regions, which are easy to implement as only a single borderline between regions has to be moved. The cumulative effect of a sufficiently large number of such swaps is that they end up in what could be deemed as a *random* permutation. The new binary variant we suggest in this work is to decide according to the current bit of the secret key, whether to perform the swap of a given interval with its left or right adjacent neighbor, working cyclically for the extreme intervals.

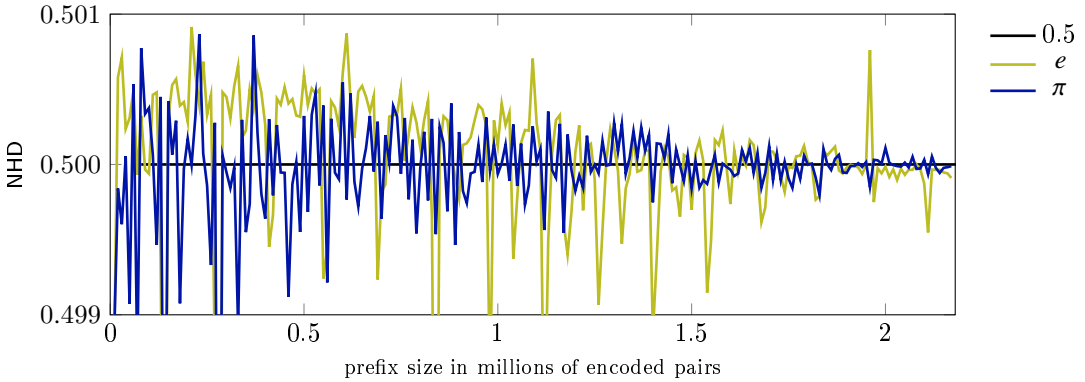

Figure 2. NHD for two runs on the same text with different keys for arithmetic coding.

The idea is similar to that of the Huffman coding seen in the previous section: even if an eavesdropper knows about the strategy of constantly swapping adjacent intervals, decoding will not be possible without knowing the side of the swap, and guessing it is impossible. Figure 2 illustrates the NHD of the encrypted file with one produced without this encryption, again for the two seeds e and π used above. Here the convergence to the expected limit $\frac{1}{2}$ is even faster (note the scale on the y-axis), as expected for arithmetic coding, whose output, even without encryption, is known to be especially close to random [7].

To verify that the convergence to $\frac{1}{2}$ is not due to the difference of the expansions of the irrational numbers e and π, but rather to the random fluctuations introduced by the swapping process, we repeated the above experiments for both Huffman and arithmetic coding using almost identical secret keys: the binary expansion of e on the

one hand, and the same string, in which we have flipped the tenth bit, on the other hand. As can be seen in Table 2, the produced files are still as different as if they were randomly generated, even though they follow both almost the same perturbation sequences.

	Huffman	arithmetic
Bible	0.5009	0.49979
English	0.5005	0.49994
DNA	0.4998	0.49995

Table 2: *Limit values of NHD comparing runs with secret keys differing only in one bit.*

An alternative way to apply small perturbations to arithmetic coding, and in fact also to other compression methods, is to tell *small lies*, that is, not always transmitting the true values. The idea here is to occasionally, instead of entering the intended sub-interval, use a predetermined *other* one, that may be the one next in size or the one adjacent in terms of character encoding. The choice of whether or not to do so is again solely guided by the bits of the secret key.

It is known that if some sort of mistake took place during the transmission of an arithmetically encoded message, this will not only result in a wrong character being perceived by the decoder in the place where the error has occurred, but also all the subsequent characters are irreversibly lost as well. And so, if an evil listener were to intercept the transmission of such an encrypted message, where we intentionally lead him into the wrong sub-intervals every so often, he would not be able to decode any significant part of the message. The opponent would need to guess correctly at every single letter if it was flipped, otherwise the decoded message will very quickly turn into random noise.

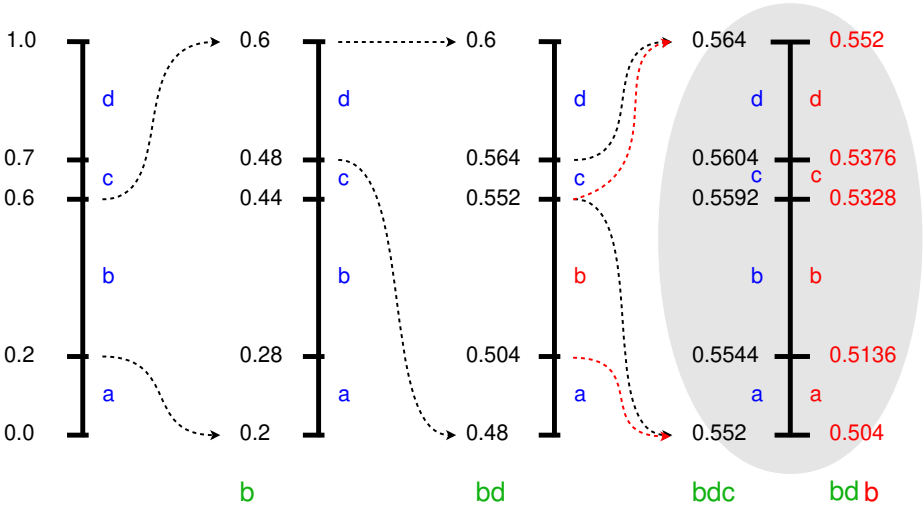

Figure 3: *Example of arithmetic coding. Each bar is a refinement of the chosen sub-interval to its left. In the rightmost step, we should have used the (black) sub-interval corresponding to character* c*, but we use the (red) sub-interval instead, corresponding to* b*. This has as effect that the partition process continues with the red partition, written to the right of the bar, instead of with the black partition, written to its left.*

An illustration of such a transmitted *lie* is given in Figure 3, depicting a simple example with an alphabet of four letters {a, b, c, d}, appearing with probabilities 0.2, 0.4, 0.1 and 0.3, respectively. Suppose the message to be encoded starts with bdc$\cdots$.

The first two steps recall the mechanism of arithmetic coding, narrowing the initial interval $[0, 1)$ first to $[0.2, 0.6)$ according to the first character b, and then further to $[0.48, 0.6)$ for the second character d. In the regular process, the third character c would then yield the interval $[0.552, 0.564)$, but if we instead decide to choose the adjacent interval corresponding to b, the new interval would be $[0, 504, 0.552)$; the new partitions are depicted in black and red in the left and right parts of the shaded area.

An unauthorized decoder, who has no access to the secret key, would not know that the decoded letter b should have actually be replaced by the following letter in line instead, c in our example. Even without repeating the perturbation process, the decoding would already be mislead into a completely different path.

A possible attack would be, if it is known that the decoded output is not reliable, and that occasionally, a deliberately wrong character is transmitted, to try an exhaustive search through all the possible alternatives. This might work if only a single or very few such lies are used, but incurs a prohibitive search cost if we apply such perturbations possibly after each character.

It should be noted that the compression efficiency is necessarily hurt by introducing these lies, as opposed to the techniques mentioned before for Huffman coding, or for arithmetic coding with swapping the position of adjacent intervals, that preserve the compression optimality. This is so because the given input file defines for the characters to be encoded a probability distribution $p_1, \ldots, p_n$. By encoding from time to time different characters, we deviate from the original to a different probability distribution $q_1, \ldots, q_n$. The size of the arithmetically encoded file using this lying strategy is thus $-\sum_{i=1}^{n} p_i \log q_i$, but by a well know theorem, the size S of the original encoded file satisfies

$$S = -\sum_{i=1}^{n} p_i \log p_i \leq -\sum_{i=1}^{n} p_i \log q_i, \tag{1}$$

with equality only if $p_i = q_i$ for all i.

Pushing the idea of transmitting sporadically a different character even further, consider a strategy in which the characters of the alphabet $c_1, \ldots, c_n$ are ordered by their probabilities $p_1 \leq \cdots \leq p_n$ and in which one transmits consistently the next character with the next higher probability, that is, c_{i+1} is encoded instead of c_i for $1 \leq i < n$, and c_0 instead of c_n. This may look as a promising compression method for itself at first sight, even without any connection to encryption, because for almost all characters, there is a gain: actually using a higher probability results in narrowing less the current interval, and ultimately should require less bits for the encoding of the entire message.

Unfortunately, the most frequent character c_n being replaced by the rarest one c_1, necessarily cancels the entire gain, because of the theorem mentioned in eq. (1). The loss could even be significant. A refined solution could thus be to partition the ordered sequence of characters into several adjacency regions and to apply the cyclic shift within each region rather than globally. Formally, we define k subsequences of the n indices, 1 to n_1, $n_1 + 1$ to n_2, …, $n_{k-1} + 1$ to $n_k = n$. The shifts are then from c_i to c_{i+1} and from $c_{n_{j+1}}$ to c_{n_j+1}, for all $n_j + 1 \leq i < n_{j+1}$, with $0 \leq j < k$ and $n_0 = 0$. Table 3 presents some of the compression results for arithmetic coding on two 50MB test files from the *Pizza & Chili* corpus, with all file sizes given in MB.

	alphabet size	compressed no encryption	compressed 1 cycle	compressed 5 cycles	compressed 10 cycles
English	99	26.95	27.83	27.01	26.97
XML	97	32.69	32.99	32.79	32.72

Table 3: *Arithmetic compression performance on test files. The column headed* no compression *refers to pure arithmetic compression without the lying heuristic.*

As can be seen, while using a single cycle incurs an increase of 1–3% in the size of the compressed file, partitioning the alphabet into 5–10 equisized cycles reduces this loss to less than 0.1%.

2.3 LZW

Welch's variant of the LZ78 algorithm [20] builds a dictionary $\mathcal{D}$ that is initialized by the single characters, and grows dynamically by the addition, at each stage, of the longest newly encountered substring. The algorithm consists of two independent, yet interleaving, processes:

1. greedily parsing the given text T by finding the longest element $A \in \mathcal{D}$ matching the characters following the current position in T;
2. inserting a new element $B = Ax$ into $\mathcal{D}$, where x is the first character that caused a mismatch.

The processes are independent, because one could at any point, and in particular when the dictionary fills up, stop the updating of the second process and continue with the first alone, as in a static dictionary parsing. The idea to turn LZW into a compression-crypto system is to perform the update process of the second point selectively, according to the 1-bits of the secret key $\mathcal{K}$. This is similar to the selective update process of the statistics in dynamic arithmetic coding [8], where instead of basing the model on the last seen m characters, one chooses randomly m of the $2m$ last characters.

There is, however, a compression deterioration of 1–3% in this variant of LZW on our test files, while for arithmetic coding the fluctuations in the size of the compressed file were hardly noticeable. Figure 4 plots the NHD in the same format as above, showing again convergence to the expected $\frac{1}{2}$. As already noted in [7], the output of LZW is not fully random, since it consists of pointers of predetermined sizes (the first 256 pointers use 9 bits, the next 512 have 10 bits, etc.), and their leading bits have a higher chance to be 0 than 1. We therefore extracted the two first bits of all the codewords into a separate file, that could be compressed by arithmetic coding. The output of this combined process has been used to generate the graphs in Figure 4.

2.4 LZ77

The output of the variant LZSS [17] of the LZ77 algorithm is a sequence of items, each of which can be either a single character or an (offset, length) pair. A 1-bit flag is used to differentiate between the alternatives. In practice, after a sufficiently long prefix, the output consists only of a sequence of pairs.

We adapt the strategy of not always telling the truth by adding some small integer d to the offset, if and only if the current bit in the secret key $\mathcal{K}$ is 1. If such a disruption

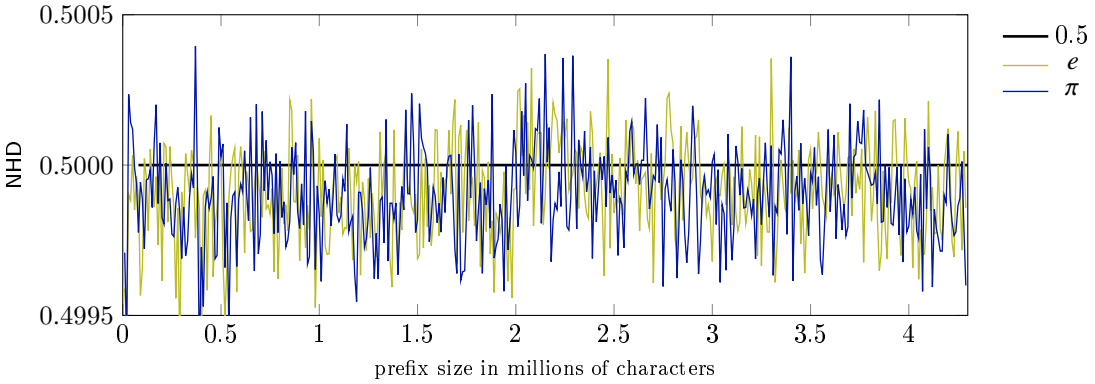

Figure 4: NHD for two runs on the same text with different keys for LZW.

occurs only once, an opponent who does not know that the offset has been shifted, will copy some wrong characters. This may seem as a local perturbation, while a single wrong bit in the output of arithmetic coding generally plays havoc with the rest of the file. However, even for LZ77, the wrong characters may be referenced later, adding more errors, and this may lead to a snowball effect, destroying eventually the message completely. The slight increase of the offsets caused a compression loss of 2-4% on our test files. The NHD graphs also show convergence to the expected $\frac{1}{2}$ and are omitted.

An additional twist to complicate malicious decoding attempts even further is to let the shift parameter d also depend on $\mathcal{K}$: one could, for instance, reuse four of the last processed bits of the key and define d accordingly as an integer between 1 and 16.

2.5 Burrows-Wheeler Transform

The Burrows-Wheeler transform is not a compression method on its own and only permutes its input. It has, however, a tendency to produce long runs of identical characters, and its output is generally much more compressible, which is why it is cascaded with techniques like Move-to-front (MTF) followed by run-length-encoding in popular software like bzip2.

Previous work with the BWT includes Külekci [9], who proposes a compression cryptosystem based on a random permutation of the alphabet in the BWT; the system supports pattern matching on the compressed form, while still retaining its security. Similar alterations to the BWT are mentioned in Teuhola [11] with applications to clustering.

Our suggestion here is to decide according to $\mathcal{K}$ whether to apply MTF or its variant Move-to-middle (MTM). Indeed, both heuristics are plausible: MTM reacts slower to dramatic changes in the text, but might be preferable for sporadic appearances of rare characters. On our test files, MTM gave slightly lower compression, and the suggested MTF+MTM approach yielded an increase of up to 8% in size, and similar NHD graphs.

3 Conclusion and future work

We presented techniques to turn some of the classical compression methods into crypto-systems, based on a secret key shared by sender and receiver, at the cost of

a mostly negligible loss in compression efficiency. Of course, the randomness alone of the output does not imply that the systems are secure against attacks and we shall work on this aspect, for example by trying to show the NP-completeness of the problem of breaking the code, as done in [3] or [5].

References

1. N. Brisaboa, S. Ladra, and G. Navarro: *DACs: Bringing direct access to variable-length codes.* Inf. Process. Manag., 49(1) 2013, pp. 392–404.
2. B. Carpentieri: *Efficient compression and encryption for digital data transmission.* Security and Communication Networks, 9591768 2018, pp. 1–9.
3. A. Fraenkel and S. Klein: *Complexity aspects of guessing prefix codes.* Algorithmica, 12(4) 1994, pp. 409–419.
4. Y. Gluck, N. Harris, and A. Prado: *Breach: Reviving the crime attack.* Black Hat Conference, Las Vegas, USA, July 27–August 1, 2013.
5. Y. Gross, S. Klein, E. Opalinsky, R. Revivo, and D. Shapira: *A Huffman code based crypto-system*, in Data Compression Conference, DCC'22, Snowbird, UT, USA, March 22-25, 2022, pp. 133–142.
6. J. Kelley and R. Tamassia: *Secure compression: Theory & Practice.* IACR Cryptol. ePrint Arch., 2014, p. 113.
7. S. Klein and D. Shapira: *On the randomness of compressed data.* Inf., 11(4) 2020, p. 196.
8. S. Klein and D. Shapira: *Integrated encryption in dynamic arithmetic compression.* Inf. Comput., 279:104617 2021.
9. M. O. Külekci: *On scrambling the Burrows-Wheeler transform to provide privacy in lossless compression.* Comput. Secur., 31(1) 2012, pp. 26–32.
10. A. Moffat: *Word-based text compression.* Softw. Pract. Exp., 19(2) 1989, pp. 185–198.
11. A. Niemi and J. Teuhola: *Burrows-Wheeler post-transformation with effective clustering and interpolative coding.* Softw. Pract. Exp., 50(9) 2020, pp. 1858–1874.
12. J. Raju: *A study of joint lossless compression and encryption scheme*, in International Conference on Circuit, Power and Computing Technologies, IEEE, 2017, pp. 1–6.
13. N. Sangwan: *Combining Huffman text compression with new double encryption algorithm*, in C2SPCA Conference, IEEE, 2013, pp. 1–6.
14. E. Setyaningsih and R. Wardoyo: *Review of image compression and encryption techniques.* Intern. J. of Advanced Computer Science and Applications, 8(2) 2017, pp. 83–94.
15. R. Sharma and S. Bollavarapu: *Data security using compression and cryptography techniques.* International J. of Computer Applications, 117(14) 2015.
16. A. Singh and R. Gilhotra: *Data security using private key encryption system based on arithmetic coding.* International Journal of Network Security & Its Applications (IJNSA), 3(3) 2011, pp. 58–67.
17. J. Storer and T. Szymanski: *Data compression via textual substitution.* J. ACM, 29(4) 1982, pp. 928–951.
18. T. Subhamastan Rao, M. Soujanya, T. Hemalatha, and T. Revathi: *Simultaneous data compression and encryption.* International Journal of Computer Science and Information Technologies, 2(5) 2011, pp. 2369–2374.
19. C. Wang: *Cryptography in data compression.* Code-Breakers Journal, 2(3) 2006.
20. T. Welch: *A technique for high-performance data compression.* IEEE Computer, 17(6) 1984, pp. 8–19.
21. K. Wong, Q. Lin, and J. Chen: *Simultaneous arithmetic coding and encryption using chaotic maps.* IEEE Trans. on Circ. and Syst.–II Express Briefs, 57(2) 2010, pp. 146–150.
22. K. Wong and C. Yuen: *Embedding compression in chaos based cryptography.* IEEE Trans. on Circuits and Systems–II Express Briefs, 55(11) 2008, pp. 1193–1197.
23. D. Zebari, H. Haron, D. Zeebaree, and A. Zain: *A simultaneous approach for compression and encryption techniques using deoxyribonucleic acid*, in 2019 13th SKIMA Conference, IEEE, 2019, pp. 1–6.