



A Huffman Code Based Crypto-System

Yoav Gross*  Shmuel T. Klein**  Elina Opalinsky*

Rivka Revivo* Dana Shapira* 

*Ariel University

Dept. of Computer Science

Ariel 40700, Israel

{yodgimmel, rivkarevivo1997}@gmail.com

elinao@ariel.ac.il, shapird@g.ariel.ac.il

**Bar Ilan University

Dept. of Computer Science

Ramat-Gan 52900, Israel

tomi@cs.biu.ac.il

Abstract

A Compression Cryptosystem is a single coding process, the output of which is both reduced in space and secure against unauthorized decoding. Considering Huffman coding, this paper proposes to apply repeatedly minor changes to the compression model, with negligible deterioration of its optimality. The cumulative impact of a large number of such changes leads to completely different ciphertexts, which can be decoded only if a given secret key is known. The security of the system is based on the NP-completeness of a problem related to breaking the code. Several variants are suggested, and their results are tested in various settings, including for security against chosen plaintext attacks.

1 Background

Transmitting information over a communication network is challenged by processing rate, increasing throughput, and by the possibility to protect the underlying data. With the increasing number of cyber-attacks and threats, these goals become all necessary when dealing with enormous amounts of delicate information. Combining compression and encryption methods can overcome these challenges by representing the data compactly and in a secure format. Compression cannot be applied on encrypted data, because the latter usually cannot be distinguished from a randomly generated file, as empirically shown by Carpentieri [1] and by Sharma and Bollavarapu [9]. Therefore, when both compression and encryption are desired, compression must be applied before encryption or simultaneously.

Simultaneous compression and encryption, forming so-called *Compression Cryptosystems*, can be achieved by either embedding compression into encryption algorithms as in [15], or by adding cryptographic features into the compression scheme as in [4], and as we suggest in this paper. We devise a single process, the output of which is both reduced in space and secure against unauthorized decoding. Contrary to the method in [4], in which this was achieved by controlling the coding *model* by means of a secret key, the present work controls the code generation itself. Moreover, while the method of [4] cannot *guarantee* the same compression ratio (though in practice,

it achieves it), for the basic new method proposed herein, the compression is not affected.

The rationale behind the proposed algorithm is to provide encryption while retaining the same compression ratio as without encryption. That is, the compression cryptosystem maintains the same model and distributions as the compression method it is based on, while using the secret key to control which of the many possible encodings to be chosen. Decoding the ciphertext is infeasible without the exact knowledge of the key, and guessing a long enough random key is obviously ruled out. Although simultaneous compression and encryption has already been studied, most of these schemes are found to be insecure and especially inefficient in terms of compression.

A comprehensive survey regarding the combination of cryptographic and compression techniques is given by Setyaningsih and Wardoyo [8], considering both lossy and lossless compression methods and mainly focusing on image compression. The authors note that most of the studies concentrate more on image security than on data size reduction. Wang [13] applies a random shuffle of the source alphabet to several compression algorithms according to a given secret key, in a preprocessing stage. In particular, the initial *dictionary* is shuffled in LZW compression, the *order* of the alphabet symbols in the working interval is shuffled in case of arithmetic coding [14], 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 [11]. However, the effectiveness of the security is not proven in any of these works and they do not cope with *Chosen Plaintext Attacks* (CPA).

In fact, any compressed file can be encrypted by a *One Time Pad* (OTP), that is, XORing the compressed message with a one-time private key of length at least the message size. Singh et al. [10] use OTP with arithmetic coding. Empirical tests show that the processing times of simultaneous compression and encryption are shorter than applying them sequentially. Raju [6] has examined it for OTP with arithmetic coding, while Sangwan [7] tested it for a particular algorithm that combines reversed static Huffman codewords with different secret keys. A recent paper [16] converts a given plaintext into an equivalent DNA file based on a secret mapping, followed by compressing the data and then encrypting it by an additional scrambling round.

Our approach is to apply repeatedly minor changes to the compression model, without hurting its optimality. The cumulative impact of a large number of such changes leads to completely different ciphertexts, which can be decoded only if the secret key is known. In a first stage of our new method, we shall concentrate on static Huffman coding since the intuition is clearer. In order to cope with the vulnerable beginning of the output ciphertexts, we then extend our methods to their adaptive versions.

Our paper is constructed as follows. We start with a simplified compression cryptosystem based on static Huffman coding in Section 2. Section 3 addresses the security of the crypto-system. In Section 4 we extend the basic version in order to cope with CPA attacks, and show that just a small number of inserted don't-care characters can be used to counteract such attacks. Empirical results are reported in Section 5.

2 Static Huffman Coding

We assume that the input file has already been partitioned into elements to be encoded, which could be characters or words or other strings. This partition process yields a probability distribution for the set of elements, and we shall assume it to be fixed in this section. An internal node v of the corresponding Huffman tree is selected according to a secret key, which is assumed to have been exchanged between the encoder and decoder prior to the coding phases. Any node v of a Huffman tree (or in fact, the tree of any prefix code) is associated with the string obtained by concatenating the labels on the edges on the path from the root to v , where left and right edges are labelled 0 and 1, respectively. Some *transformation* is then applied on the subtree rooted by the selected node v , making sure that all the codeword lengths remain the same, so that, unlike for previous research, our algorithm preserves the optimal compression ratio of Huffman encoding the original file. The cumulative impact of several consecutive such transformations is to produce completely different output files, yet all of optimal minimal length. We consider several possibilities for applying a *transformation* at an internal node v :

1. *A mirror transformation:* The whole tree rooted at v is replaced by its mirror image. As example, consider the original tree given in Figure 1(a), and the gray node as the selected internal node v . Its mirror transformation is shown in Figure 1(b). As a result, the codewords corresponding to leaves of the tree rooted at v are transformed from $\alpha\beta$ to $\alpha\bar{\beta}$, where α is the string corresponding to v , β is the suffix of a specific codeword before the swap, and $\bar{\beta}$ is the 1's complement of β . In our example, the chosen node v corresponds to $\alpha = 1$ and the codeword 10110 assigned to leaf i is transformed into $\overline{10110} = 11001$.
2. *A swap transformation:* Interchange only the left and right subtrees of the internal node v , without continuing recursively to the subtrees as done in the mirror variant above. On our example, the original Huffman tree is transformed into Figure 1(c). That is, the codeword $\alpha b\gamma$ is converted to $\alpha\bar{b}\gamma$, where α and γ are strings and b is a single bit.

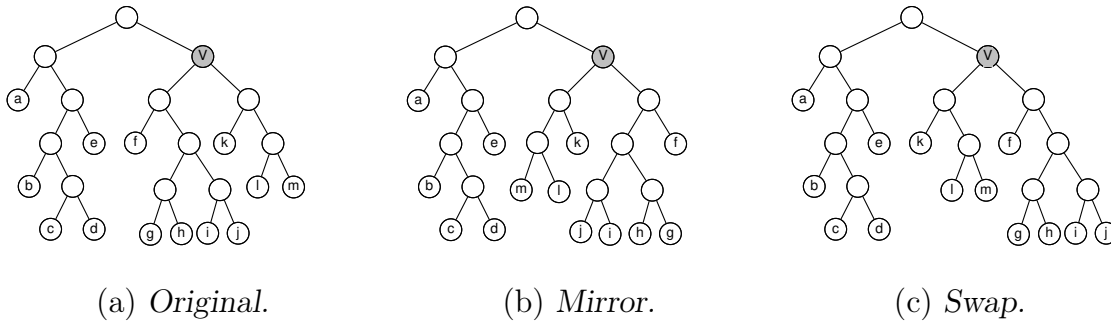


Figure 1: Mirror and swap Transformations

Both suggested transformations can be applied any number of times, separately or combined, and even selectively according to an additional bit of the secret key, in order to achieve a better shuffling of the encoded bits and thereby better encryption for the model, while preserving the lengths of all codewords.

Given a text $T = x_1x_2 \cdots x_n$ to be compressed and encrypted, where the x_i are characters belonging to some alphabet Σ of size σ , the CRYPTO-HUFFMAN encoding algorithm uses initially the Huffman tree of a static encoding. In fact, in our experiments we started with a Huffman tree \mathcal{T} that was constructed by a non-deterministic process. In a deterministic implementation, when processing pairs of weights, the smaller weight may always be assigned to the same side (left or right subtree of the current node), which may bias the overall distribution of the bits using this encoding. The non-deterministic creation of the Huffman tree tends to balance the occurrences of these 0 and 1-bits, as it can also be seen as using an initialization process that processes the internal nodes of a static Huffman tree, deciding randomly whether or not to swap the subtrees of the following node.

An additional parameter of the encryption process is an integer k , chosen in the range $1 \leq k < \sigma$. After a character x_i has been read, the corresponding codeword is output to the encoded file. Then, k different internal nodes $\{v_1, \dots, v_k\}$ are chosen according to a secret key, and a *transformation* on the subtree of \mathcal{T} rooted at node v_i is applied, for $1 \leq i \leq k$. The objective of the parameter k is giving the ability to control the trade-off between security and time complexity.

Alternatively, instead of explicitly pointing to the k internal nodes on which the transformations should be applied, which requires $k \log \sigma$ bits from the secret key, we may just use $\sigma - 1$ bits if this is smaller, and indicate at each internal node whether or not to apply the transformation. Use the $\sigma - 1$ following bits of the secret key to refer to the internal nodes of \mathcal{T} in some order. The *transformation* will be applied only to those internal nodes that are paired with 1-bits. We call this variant BULK-CRYPTO-HUFFMAN.

The mirror and swap transformations might be easy to implement, as basic operations, but they can only produce $2^{\sigma-1}$ permutations for the σ leaves of the tree, which is much less than the $\sigma!$ possible permutations. In fact, the number of permutations to be considered depends on the shape of the optimal tree. For a full tree, having all σ leaves on the same level, there are indeed $\sigma!$ permutations. On the other hand, if the optimal tree turns out to be degenerate, that is, with one leaf on level i for $1 \leq i < \sigma - 1$, and two leaves on the lowest level $\sigma - 1$, then only $2^{\sigma-1}$ permutations are possible.

The following operation improves the situation (though not in the extreme case of a degenerate tree) and generalizes the two extreme cases above by applying swaps to any pair of adjacent nodes within the same level, and not only siblings. The transformation still retains the codeword lengths, thus the compression efficiency does not get hurt. Formally,

3. *A level-swap transformation:* Consider all the nodes, leaves and internal, within the same level i , for $i > 0$, as belonging to a cyclic linked list. Interchange

subtrees rooted at a node v with the node $right(v)$ to its right, on the same level. In particular, the rightmost node is exchanged with the leftmost. The codewords $\alpha_1\beta$ and $\alpha_2\gamma$ are transformed into $\alpha_2\beta$ and $\alpha_1\gamma$, respectively, where α_1 and α_2 are the strings corresponding to v and $right(v)$ so that $|\alpha_1| = |\alpha_2| = i$, and β and γ are suffixes of specific codewords corresponding to leaves in the subtrees rooted by v and $right(v)$ before the swap.

When combining k level-swap transformations, the resulting Huffman tree depends on the order the k nodes have been selected. Performing the same operations in a different order is likely to produce different trees, in contrast to the first two transformations in which any order of the same selected k nodes results in the same output Huffman tree. Consequently, selecting the k vertices using $k \log \sigma$ bits strengthens the encryption, as compared to a single level-swap, more than using the same amount of bits for k swap and mirror operations.

3 Security and Resistance to Cipher Attacks

Given is a text $T = x_1x_2 \cdots x_n$, the encoder produces the corresponding ciphertext C , which is a binary sequence obtained by applying a prefix encoding to the elements x_i of T . That is, there is a partition of C into codewords $c(x_i)$, satisfying the **Prefix property** and **Consistency** in the sense that

$$\forall \ 1 \leq i, j \leq n \quad c(x_i) = c(x_j) \Leftrightarrow x_i = x_j.$$

The objective of encryption is to hide the content of a given plaintext file from an unauthorized eavesdropper. The goal of such an opponent, on the other hand, is to try to *break the code*, that is, reveal the function $c()$. This can then be used not just to decipher the currently given secret plaintext, but also all subsequently generated input files, if they are encrypted by means of the same encoding function.

Decryption attempts can be based on statistics of the occurrences of the alphabet symbols in natural languages, which are generally well-known. These statistics may lead to guessing the approximate codeword lengths with high probability, thereby inducing a much smaller number of possible partitions of the ciphertext into codewords, so that exhaustively checking all such partitions can not be ruled out any more. To counteract such attacks, [3] propose to insert random bits at the end of certain codewords, which is cryptographically secure in the sense that the problem of guessing the code in such a setting is shown to be NP-complete.

Our suggestion of continuously modifying the shape of the Huffman tree by applying swaps or similar transformations can be seen as an alternative attempt to introduce enough perturbations into the ciphertext to turn the problem of breaking the code into a difficult one. We assume, as usual, that the opponent has knowledge of all the details of the encryption process, which are the various swap, mirror or other transformations, to be applied at various locations within the Huffman tree. The only hidden parts are those depending on the secret key that controls when, where and

which of these transformations are used. Since guessing the key may be ruled out, the question is whether it is possible to derive any details of the encoding function $c()$ by other means.

Each of the suggested transformations turns the current set of codewords into a different one, albeit without changing their lengths, so that the optimality of the Huffman code is kept. The ciphertext thus consists of the concatenation of codewords belonging to different codes, thus the **consistency** condition mentioned above will not hold any more. By repeatedly applying several different such transformations, the expected effect is to constantly alter the shape of the Huffman tree and thereby the set of used codewords, which should prevent the opponent from finding the proper partition of the ciphertext into codewords.

To argue that breaking the code in such a setting is probably not feasible in reasonable time, we prove in the full version of this paper that the following related problem is NP-complete.

Swap Mirror Prefix Code (SMPC):

Input: Given are positive integers ℓ , k and p , a text $T = x_1x_2 \cdots x_n$ over some alphabet Σ of size $\sigma = |\Sigma|$, an initial Huffman tree H for Σ , a set of k transformations of the type **swap** or **mirror** as explained above, and a binary sequence S .

Question: Is there a subsequence S' of S of length $|S'| = \ell$, such that S' can be partitioned into codewords of the original Huffman code induced by H , the partition satisfying that

1. the lengths of the codewords belong to $\{s, \dots, s + p\}$ for some integer s ;
2. the set of these codewords satisfies the prefix property;
3. each codeword in S' consistently encodes a character of T ;
4. one of the k transformations is applied to the current Huffman tree after the processing of some of the characters.

Note that we assume no relationship between the parameters ℓ , k , p and n .

4 Chosen plaintext attacks

In order to cope with CPA attacks, we define a new symbol DC, which will act as a *don't-care*, and adjoin it to the alphabet to yield $\Sigma' = \Sigma \cup \{\text{DC}\}$. The goal is to produce different ciphertexts, even in case the same secret key is used to encrypt the same message. This DC symbol is repeatedly inserted to the input file at randomly chosen positions.

There seems to be an inherent weakness in the crypto-security of using static Huffman encoding, even in the presence of randomly inserted DCs. If we consider two encoding attempts of the same text and using the same secret key, and let only the positions differ where the DCs are inserted, we shall get identical output on all ranges for which the number of inserted DCs so far are equal for the two encodings. This weakness could be exploited by an opponent for a crypto-attack. Note that the matching parts disappear as soon as we turn to *dynamic Huffman* coding [12], because the codewords assigned to the different characters constantly change.

We still have to decide about the number of DC symbols that should be inserted. This number should balance between the effort of not increasing the text significantly, and the fact that with a too low number of DCs, the file is prone to attacks. We therefore suggest the following tradeoff.

The insertion of the DC symbols will be according to a stochastic process controlled only by the encoder, and independent of the secret key \mathcal{K} . Such a setting is possible, because the decoder does not need to know in advance where the DCs are inserted, and will recognize them after they are decoded. At position i in the text to be generated, $1 \leq i \leq n$, the DC symbol is added with probability $\frac{\log(i+1)}{ci}$, where $c > 1$ is a constant controlling the total number of inserted DCs. We approximate the expected distance between successive occurrences of DCs at position i by $\frac{ci}{\log(i+1)}$, as if we were using a constant probability between successive insertions of DCs. The overall expected distance between DCs for the entire range is then $E = \frac{1}{n} \sum_{i=1}^n \frac{ci}{\log(i+1)} = \theta(\frac{n}{\log n})$. To see this, we bound E on both sides:

$$E \geq \frac{1}{n} \sum_{i=1}^n \frac{ci}{\log(n+1)} = \frac{c}{n \log(n+1)} \sum_{i=1}^n i = \frac{c(n+1)}{2 \log(n+1)} = \Omega\left(\frac{n}{\log n}\right).$$

On the other hand,

$$\begin{aligned} E &\leq \frac{1}{n} \sum_{i=1}^{\sqrt{n}} \frac{ci}{\log(i+1)} + \frac{1}{n} \sum_{i=\sqrt{n}+1}^n \frac{ci}{\log(i+1)} \leq \frac{c}{n} \sum_{i=1}^{\sqrt{n}} i + \frac{c}{n} \sum_{i=\sqrt{n}+1}^n \frac{n}{\log(\sqrt{n})} \\ &= \frac{c \sqrt{n}(\sqrt{n}+1)}{2n} + \frac{c}{n} (n - \sqrt{n}) \frac{2n}{\log n} = O\left(\frac{n}{\log n}\right). \end{aligned}$$

Thus the expected number of DCs, $\theta(\log n)$, is not bounded, and the *fraction* $\frac{\log n}{n}$ of inserted elements tends to zero.

In addition, each insertion of a DC into the text will also affect the Huffman tree, but differently than the other characters. While after each new character, a swap or mirror transformation is applied to the Huffman tree, the action taken after a DC is to skip over a small constant number h of bits of the secret key. Such a choice sets a bound of $\theta(\log n)$ for the expected total number of bits wasted from the secret key, which is about the order of the number of bits used for the transformations on the Huffman tree induced by just a few characters of the text. On the other hand, this seems immune even to CPA attacks, because even if the encoding is applied on the same file and with the same secret key \mathcal{K} , the inserted DCs are likely to be in different positions and will thus produce different disturbances. The formal description is presented in Algorithm 1. The input also includes the parameter h for the number of bits to be skipped in case of a DC. Decryption is done by ignoring all DCs.

To remain with the same model in different runs of the encoding algorithm when applied on the same plaintext, the sequences of locations where the DCs are inserted have to be identical, because even a single discrepancy will shift the selected bits of the

Algorithm 1: Generic Crypto-Compression — Encoding

CRYPTO-COMPRESSION-ENCODE($x_1x_2 \cdots x_n, k, \mathcal{K}, h$)

- 1 initialize the model
- 2 **for** $i \leftarrow 1$ **to** n **do**
- 3 choose randomly a probability value p
- 4 **if** $p < \frac{\log(i+1)}{c i}$ **then**
- 5 encode DC according to the current model
- 6 skip h bits in \mathcal{K}
- 7 encode x_i according to the current model
- 7 use the secret key \mathcal{K} to select k different symbols in Σ
- 7 **for** $j \leftarrow 1$ **to** k **do**
- 8 apply *transformation* on the model

secret key \mathcal{K} controlling the transformations. However, the probability of producing identical insertion sequences after r processed characters, $\prod_{i=1}^r \frac{\log(i+1)}{c i} < \frac{1}{2^{r-j}}$ for some constant j , decreases exponentially with r , and therefore, the probability of two different runs to coincide rapidly tends to zero.

5 Empirical Tests

For our experimental results we considered the *Large Corpus* taken from the Canterbury¹ corpora. As the results are quite typical for all files, we here only present the outcome for *bible.txt*, the King James version of the Bible, of size 4,047,392 Bytes.

As a first test of security, any reasonably compressed or encrypted file should consist of a sequence of bits that is not distinguishable from a randomly generated binary sequence. A criterion for such randomness could be that the probability of occurrence, within the compressed file, of any substring of length m bits should be 2^{-m} , for all $m \geq 1$, that is, the probability for 1 or 0 are both 0.5, the probabilities for 00, 01, 10 and 11 are 0.25, etc. We checked this fact for values of m up to 3 on our encrypted output files for CRYPTO-HUFFMAN, using the mirror and swap transformations with k equals 1 or 2, and for the BULK-CRYPTO and LEVEL-SWAP variants. We repeated our experiments for both static and dynamic variants, and some of the results are presented in Table 1 (static in grey, dynamic in black). As can be seen, static Huffman, even though applied on an *a priori* randomizing, non-deterministic Huffman tree, is the method whose distribution is the most distant from uniform, while the closest are BULK-CRYPTO and LEVEL-SWAP, denoted by BULK and L-SWAP for short. To measure the deviation of the given distribution of the 2^m possible bit patterns of length m from the expected uniform distribution also for larger m , we used the Kullback–Leibler divergence [5], and got similar results, which are omitted for lack of space.

¹<http://corpus.canterbury.ac.nz>

bit-string	STATIC	L-SWAP	MIRROR-1	MIRROR-2	SWAP-1	SWAP-2	BULK
0	0.495477	0.500097	0.500061	0.500091	0.500138	0.500092	0.500034
1	0.504523	0.499903	0.499939	0.499909	0.499862	0.499908	0.499966
00	0.249532	0.250076	0.249985	0.250030	0.250074	0.250030	0.250060
01	0.245944	0.250021	0.250076	0.250060	0.250074	0.250060	0.249974
10	0.245944	0.250021	0.250076	0.250060	0.250074	0.250060	0.249974
11	0.258579	0.249882	0.249863	0.249850	0.249798	0.249849	0.249992
000	0.122962	0.121652	0.124964	0.124989	0.125050	0.124989	0.125053
001	0.126570	0.128424	0.125022	0.125042	0.125024	0.125042	0.125007
010	0.119103	0.121522	0.125012	0.125093	0.124975	0.125093	0.124968
011	0.126842	0.128499	0.125063	0.124968	0.125089	0.124968	0.125007
100	0.126570	0.128424	0.125022	0.125042	0.125024	0.125042	0.125007
101	0.119374	0.121597	0.125054	0.125018	0.125040	0.125018	0.124967
110	0.126842	0.128499	0.125063	0.124968	0.125089	0.124968	0.125007
111	0.131737	0.121384	0.124800	0.124882	0.124709	0.124882	0.124985

TABLE 1: Probability of 1-, 2- and 3-bit substrings for the Dynamic Huffman variants.

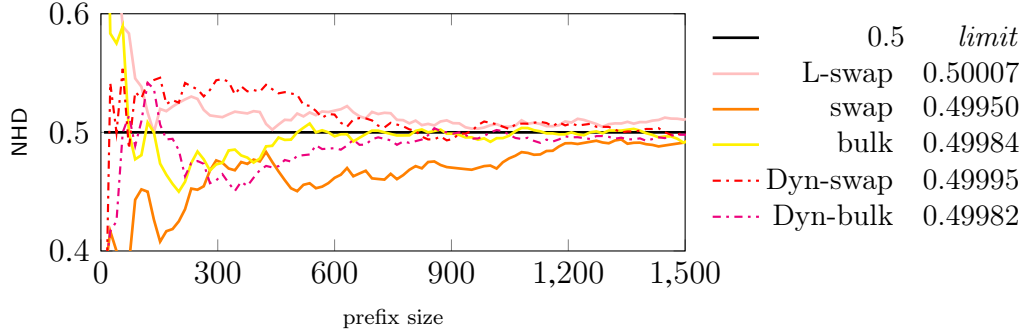


Figure 2: NHD for two runs on the same text with the same key, but different DCs.

We also wanted to check the sensitivity of the system to variations in the secret key. As we consider encrypted files, we may use the *normalized Hamming distance* as measure, which is in fact adapted from the measure of similarity between two image files proposed in [2], and has been defined as the normalized number of differing pixels. Given two bitstrings $A = a_1 \cdots a_n$ and $B = b_1 \cdots b_m$ with $n \geq m$, B is first extended by zeros so that both strings are of the same length n . The normalized Hamming distance is then defined as

$$\text{NHD}(A, B) = \frac{1}{n} \sum_{i=1}^n (a_i \text{ XOR } b_i).$$

We examined a pair of ciphertexts, generated according to two independently chosen random keys, for each of our suggested encoding variants. The produced ciphertexts were completely different, with the weighted number of differences in corresponding bits rapidly tending to the expected value $\frac{1}{2}$. The limiting values obtained after processing the entire input file were, e.g., 0.50008 for BULK-CRYPTO and 0.49983 for DYNAMIC-SWAP. We conclude that the proposed crypto-compression

procedure is extremely sensitive to even small alterations: the produced files pass the above randomness tests, are practically of the same size, and are completely different from each other, while preserving the compression efficiency.

In order to evaluate the resistance of the proposed compression cryptosystem to CPAs, we computed the normalized Hamming Distance between two encodings of the *same* text, using the *same* key and only differing in the positions where the DC elements have been inserted. There is again convergence towards the expected limit $\frac{1}{2}$, at the price of a negligible hurt in compression efficiency as can be seen in Figure 2.

Acknowledgement: The authors would like to thank the Israeli Innovation Authority and Defender Cyber Technologies LTD for their support for research under the Academic Knowledge Guidance Program (Nufar) File No. 69133 + 69098.

References

- [1] B. Carpentieri. Efficient compression and encryption for digital data transmission. *Security and Communication Networks*, 2018:9591768:1–9591768:9, 2018.
- [2] L. Duan, X. Liao, and T. Xiang. A secure arithmetic coding based on Markov model. *Comm. in Nonlinear Science and Numerical Simulation*, 16(6):2554–2562, 2011.
- [3] A. S. Fraenkel and S. T. Klein. Complexity aspects of guessing prefix codes. *Algorithmica*, 12(4):409–419, 1994.
- [4] S.T. Klein and D. Shapira. Integrated encryption in dynamic arithmetic compression. *Inf. Comput.*, 279:104617, 2021.
- [5] S. Kullback and R.A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.
- [6] J. Raju. A study of joint lossless compression and encryption scheme. In *International Conference on Circuit, Power and Computing Technologies*, pages 1–6. IEEE, 2017.
- [7] N. Sangwan. Combining Huffman text compression with new double encryption algorithm. In *C2SPCA Conference*, pages 1–6. IEEE, 2013.
- [8] E. Setyaningsih and R. Wardoyo. Review of image compression and encryption techniques. *Intern. J. of Advanced Computer Science and Applications*, 8(2):83–94, 2017.
- [9] R. Sharma and S. Bollavarapu. Data security using compression and cryptography techniques. *International J. of Computer Applications*, 117(14), 2015.
- [10] 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):58–67, 2011.
- [11] 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):2369–2374, 2011.
- [12] J.S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM (JACM)*, 34(4):825–845, 1987.
- [13] C.-E. Wang. Cryptography in data compression. *Code-Breakers Journal*, 2(3), 2006.
- [14] 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):146–150, 2010.
- [15] K. Wong and C. H. Yuen. Embedding compression in chaos based cryptography. *IEEE Trans. on Circuits and Systems–II Express Briefs*, 55(11):1193–1197, 2008.
- [16] D.A. Zebari, H. Haron, D.Q. Zeebaree, and A.M. Zain. A simultaneous approach for compression and encryption techniques using deoxyribonucleic acid. In *2019 13th SKIMA Conference*, pages 1–6. IEEE, 2019.