Secure Arithmetic Coding

Hyungjin Kim, *Student Member, IEEE*, Jiangtao Wen, *Senior Member, IEEE*, and John D. Villasenor, *Senior Member, IEEE*

***Abstract*—Although arithmetic coding offers extremely high coding efficiency, it provides little or no security as traditionally implemented. We present a modified scheme that offers both encryption and compression. The system utilizes an arithmetic coder in which the overall length within the range [0,1) allocated to each symbol is preserved, but the traditional assumption that a single contiguous interval is used for each symbol is removed. Additionally, a series of permutations are applied at the input and the output of the encoder. The overall system provides simultaneous encryption and compression, with negligible coding efficiency penalty relative to a traditional arithmetic coder.**

***Index Terms*—Arithmetic codes, cryptography, data compression.**

I. Introduction

ARITHMETIC coding has been developed extensively since its introduction several decades ago [1], and is notable for offering extremely high coding efficiency. While many earlier-generation image and video coding standards such as JPEG, H.263, and MPEG-2 relied heavily on Huffman coding for the entropy coding steps in compression, recent generation standards including JPEG2000 and H.264 utilize arithmetic coding [2], [3]. This has led to increased interest in arithmetic coding both in the context of image coding, and also more generally for other applications. While arithmetic coding is extremely efficient, as Cleary *et al.* [4] and others have noted, as traditionally implemented it is not particularly secure. The issue of providing both compression and security simultaneously is growing more important given the increasing ubiquity of compressed media files in a host of applications including the Internet, digital cameras, and portable music players, and the common desire to provide security in association with these files. When both compression and security are sought, one approach is to simply use a traditional arithmetic coder in combination with a well-known encryption method such as the Advanced Encryption Standard (AES). However, while this will certainly meet both goals, it fails to take advantage of the additional design flexibility and potential computational simplifications that are available if the coding and encryption are performed jointly.

Manuscript received February 28, 2006; revised August 2, 2006. This work was supported in part by the Office of Naval Research under Contract N00014-06-1-0253 and in part by the National Science Foundation under Grants CCR-0120778 and CCF-0541453. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. David J. Miller.

H. Kim and J. D. Villasenor are with the Electrical Engineering Department, University of California, Los Angeles, CA 90095 USA (e-mail: hjkimnov@ee.ucla.edu; villa@icsl.ucla.edu).

J. Wen is with Ortiva Wireless, Inc., La Jolla, CA 92037 USA (e-mail: jtwen@ortivawireless.com).

Digital Object Identifier 10.1109/TSP.2007.892710

Traditional arithmetic coding provides essentially no security in the face of a chosen plaintext attack, in which an attacker has the ability to specify a sequence of input symbols and observe the corresponding output, and to repeat this process an arbitrary number of times. For example, in a binary system with two symbols $\mathbf{A}$ and $\mathbf{B}$, it is a simple matter to choose input sequences that, in combination with their outputs, reveal the assumed probabilities of each symbol in the arithmetic coder as well as the order of the intervals. That information can then be used to decode any output from the encoder.

The issue of increasing the security of arithmetic coding has received relatively little attention in the literature. Bergen and Hogan [5] have considered the problem of inferring the underlying symbol probabilities and partitioning of the [0,1) interval using observations of an arithmetic encoder output. Liu *et al.* [6] presented a system using table-based bit sequence substitutions to provide encryption during arithmetic coding. More recently, a randomized arithmetic coding (RAC) system based on random swapping of the two intervals in a binary arithmetic coder was described by Grangetto *et al.* [7], [8], who utilized this approach to encrypt JPEG 2000 coded images. The systems in [7] and [8] modify the traditional arithmetic coder by randomly permuting the intervals in accordance with a key-generated shuffling sequence. The shuffling sequence consists of one bit per encoded symbol that determines whether the binary intervals are swapped or not when encoding that symbol. The authors of that paper were targeting applications to JPEG2000-encoded images in which a potential attacker would not have access to the original image nor be in a position to provide a particular image to be encoded. Thus, robustness to plaintext attacks was not a goal in [7] and [8]. Indeed, if an attacker of the RAC was granted access to the RAC encoder, removed from the larger JPEG2000 context for which it was designed, the number of trials needed to determine an N-bit shuffling sequence would be on the order of N, since the N output pairs corresponding to inputs that differ in exactly one symbol can be compared. Such comparisons, however, would not reveal the underlying key used to generate the shuffling sequence, so if care was taken to modify the key or avoid reinitialization of the shuffling sequence in subsequent uses of the RAC encoder, substantially higher robustness would result. In [9], we specifically consider the goal of encryption, and describe an arithmetic coding (Interval Splitting AC) approach in which the intervals associated with each symbol, which are continuous in a traditional arithmetic coder, can be split according to a key known both to the encoder and decoder. This removes the constraint that the intervals corresponding to each symbol be continuous, and instead uses a more generalized constraint that the sum of the lengths of the one or more intervals associated with each symbol be equal to its probability.

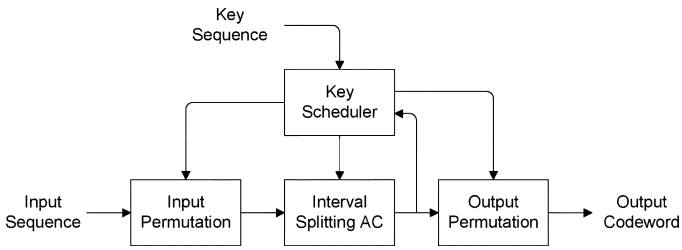


Fig. 1. Block diagram of permutation-based system.

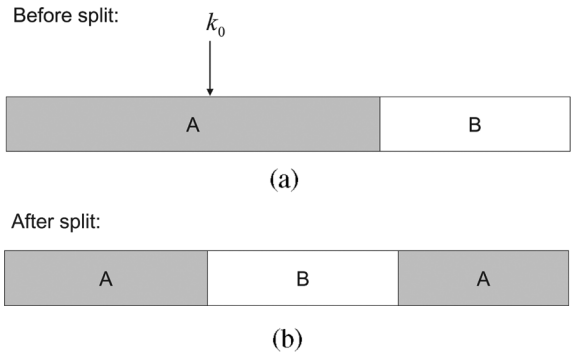


Fig. 2. Illustration of interval splitting.

The present work aims to provide an arithmetic coding system that is secure against a chosen plaintext attack. Interval splitting within the arithmetic coder and permutation steps at the input and output are utilized to construct a system with security that increases exponentially with the length of the shorter of the input block size and the key sequence. While all of the methods described here can be applied for coding of source alphabets with any size, we address the case of binary systems here to simplify the discussion and illustrations. The rest of this paper is organized as follows. Section II introduces the secure system based on interval splitting and permutation. Section III motivates the system design by describing and analyzing the contributions of each of its elements to overall security. Section IV discusses implementation complexity. Section V shows experimental results, and conclusions are contained in Section VI.

II. System Description

Fig. 1 gives a block diagram of the secure arithmetic coding system. The system consists of a first permutation step applied to the input sequence, arithmetic coding using interval splitting, and a second permutation step applied to the bits produced by the coder. A key sequence is input to a key scheduler which in turn provides key information to both permutation steps and to the interval splitting arithmetic coder. The key scheduler utilizes information from the split AC encoder output.

The permutation steps in this system are similar to the ShiftRow Transformation in AES in that the rows of a block of data are shifted cyclically. The main difference is that in the system of Fig. 1 row and column shifts are shifted cyclically by different amounts according to key values, in contrast to the corresponding step in AES in which the shifts are predetermined [10].

A. Interval Splitting Arithmetic Coder

The actual compression is performed by the interval splitting arithmetic coder. In an interval splitting AC, the intervals associated with each symbol, which are continuous in a traditional arithmetic coder, can be split according to a key known both to the encoder and decoder. For example, in a binary system with two symbols $\mathbf{A}$ and $\mathbf{B}$ and $p(\mathbf{A}) = 2/3$ and $p(\mathbf{B}) = 1/3$, a traditional partitioning would represent $\mathbf{A}$ by the range [0,2/3) and $\mathbf{B}$ by the range [2/3,1). In place of the constraint that the intervals corresponding to each symbol be continuous, the coder uses a more generalized constraint that the sum of the lengths of the one or more intervals associated with each symbol be equal to its probability. With reference to the example above, if symbol $\mathbf{A}$ is represented by the combination of the intervals [0,1/3) and [2/3,1) and symbol $\mathbf{B}$ by [1/3,2/3), the fundamental association in arithmetic coding between symbol probability and length in the range [0,1) is obviously preserved. This can be viewed as a generalization of the RAC method in [7], in the sense that it results in a coder having intervals that have been both randomly shuffled **and** split.

Traditional arithmetic coding can be viewed as an extension of Shannon–Fano–Elias coding, and associates a concatenation of N symbols x^N with prefix-free codewords of length no greater than $l(x^N) = \lceil -\log p(x^N) \rceil + 1$ bits [11]. Alternatively, if the prefix-free restriction is removed, codewords of length $\lceil -\log p(x^N) \rceil$ can be used. For a given sequence length N and input probability distribution with cdf $F(i)$, the recursive partitioning of [0,1) leads to a set of intervals $[F(i-1), F(i))$, $1 \leq i \leq 2^N$ and their associated midpoints $\bar{F}(i)$. The representation of $\bar{F}(i)$ is truncated to $l(x^N)$ bits to give $\lfloor \bar{F}(i) \rfloor_{l(x^N)}$ which is guaranteed to lie in the appropriate interval.

When an interval is split, the longer of the subintervals must be at least half as long as the presplit interval. Thus, a prefix-free representation will require at most $l(x^N) = \lceil -\log p(x^N) \rceil + 2$ bits (or $l(x^N) = \lceil -\log p(x^N) \rceil + 1$ when the prefix-free restriction is removed). In many cases, a representation shorter than these bounds is available within the longer subinterval. Alternatively and less commonly, the shortest representation will lie in the shorter subinterval. The encoder has the flexibility to pick the shortest valid codeword from either subinterval. Thus, the performance bound is one bit per N-symbol sequence larger than traditional arithmetic coding. In practice, however, the bound for interval splitting is very loose, and the average prefix-free code lengths typically range from 0.1 to 0.3 bits longer (for the entire sequence of N symbols) than the expected value of $\lceil -\log p(x^N) \rceil + 1$. Additionally, since the length increase is absolute as opposed to relative, as sequence length grows the efficiency penalty in percentage terms quickly becomes negligible [9].

Fig. 2 illustrates interval splitting for the case where only one input symbol is coded. Fig. 2(a) shows a traditional partitioning of [0,1) using an example where $p(\mathbf{A}) = 2/3$ and $p(\mathbf{B}) = 1/3$. A key k_0 can be used to identify where the interval corresponding to symbol $\mathbf{A}$ is to be split. The split causes the portion of the $\mathbf{A}$ interval to the right of the key to be moved to the right of the $\mathbf{B}$ interval as shown in Fig. 2(b). When an input containing N symbols is encoded, the key becomes a vector $k = (k_0, \ldots, k_{N-1})$ where N is the length of the input symbol string. Keys are identified using binary sequences, with the length of the binary sequence determining

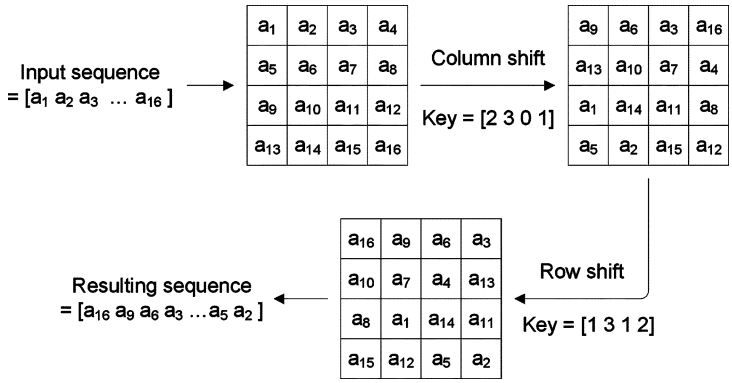


Fig. 3. Input sequence permutation ($N = 16$).

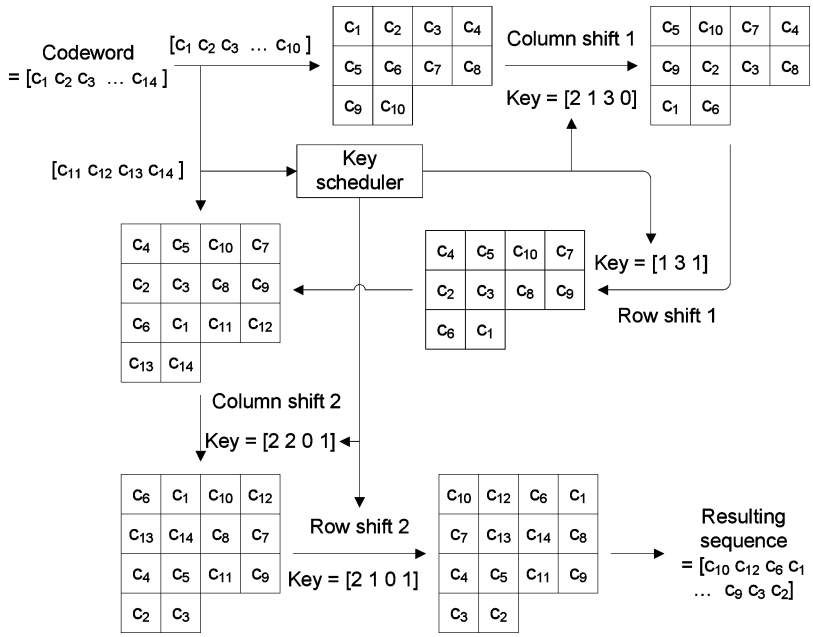


Fig. 4. Output codeword permutation ($N_c = 14$).

the precision available in key selection. For example, if three bits/symbol are utilized in key specification, the splitting will occur at one of eight possible locations. In the work here we assume that potential key locations are distributed evenly, though that constraint could be removed at the cost of some increase in complexity. Key positions are expressed in a normalized manner over the range of potential split locations. For encoding the first symbol, the key k_0 can lie anywhere on [0,1). For all subsequent symbols, however, restrictions due to previous splitting events reduce the range of eligible split locations. Thus, a key value of, for example, $k_i = 0.25$, would identify a split location at the center of the first half of the range of valid key positions associated with the ith symbol, but would not generally lie at the absolute position 0.25. Further details on the constraints involved in interval splitting and the diversity of the interval orderings that can be generated are found in [9].

B. Permutations

The input permutation operates on the symbols of the input sequence, and begins by raster-order mapping a sequence of length N into a block having four columns and $\lceil N/4 \rceil$ rows. After the mapping, two key-driven cyclical shift steps are applied, one operating on the columns and the other operating on the rows. This is illustrated for the case $N = 16$ in Fig. 3. For input sequence $(a_1, a_2, a_3, \ldots, a_N)$, the first row becomes (a_1, a_2, a_3, a_4), the second row becomes (a_5, a_6, a_7, a_8), etc. The column shifts are specified by a key of length 4, with each column undergoing a downward cyclical shift in accordance with the key value associated with that column. The values in the key lie in the range $[0, \lceil N/4 \rceil - 1]$. The procedure is then repeated, using a new key of length $\lceil N/4 \rceil$ and with values in the range [0,3], for each of the rows. The data are then read out in raster order to obtain the permuted sequence.

The output permutation operates on bits produced by the interval splitting AC. In contrast with the input permutation, which utilizes one round of cyclic shifts on the columns and then rows, the second permutation step uses two rounds as illustrated in Fig. 4. Given an output sequence of length N_c bits, the last four bits are removed to give a sequence of length $N_c - 4$. This shortened sequence is then subject to a round of key-driven column and row cyclical shifts. The last four bits are used for generating a key stream of the first round with

TABLE I
RANGES OF KEY VALUES AND KEY SIZES FOR SHIFTS IN PERMUTATIONS ($N_{c,max} = \lceil -N \log_2 p(\mathbf{B}) \rceil + 2$)

	Shifts	Ranges	Key sizes (bits)
Input Permutation	Column shift	$\left[0, \left\lceil \frac{N}{4} \right\rceil - 1\right]$	$4 \left\lceil \log_2 \left\lceil \frac{N}{4} \right\rceil \right\rceil$
	Row shift	$[0,3]$	$2 \left\lceil \frac{N}{4} \right\rceil$
Output Permutation	Column shift 1	$\left[0, \left\lceil \frac{N_{c,max}-4}{4} \right\rceil - 1\right]$	$4 \left\lceil \log_2 \left\lceil \frac{N_{c,max}-4}{4} \right\rceil \right\rceil$
	Row shift 1	$[0,3]$	$2 \left\lceil \frac{N_{c,max}-4}{4} \right\rceil$
	Column shift 2	$\left[0, \left\lceil \frac{N_{c,max}}{4} \right\rceil - 1\right]$	$4 \left\lceil \log_2 \left\lceil \frac{N_{c,max}}{4} \right\rceil \right\rceil$
	Row shift 2	$[0,3]$	$2 \left\lceil \frac{N_{c,max}}{4} \right\rceil$

the input key sequence. Therefore, if the first $N_c - 4$ bits are the same but the last four bits are different, the result of the first round is different. The last four bits of the sequence are then reappended to the resulting block, which is then subject to another round of column and row shifts. In the second round, a permutation key is a function of only input key sequence. In other words, the same key is used for all results of the first round. The result is then read out in raster order.

The reason for handling the last four bits differently is to prevent an attack from using the fact that input sequences that differ only in their last symbols will produce outputs that differ only in their final bits. In a straightforward permutation, this fact can be used to make probabilistic inferences regarding the permutation. However, since the final bits tend to differ even for very similar input sequences, applying them separately in a first round of permutation in the manner of Fig. 4 ensures that the resulting bitstreams are very different even for very similar input sequences.

Table I gives the range of key values and key sizes of each shift operation in the input and output permutation steps. The output codeword length N_c depends on the particular input sequence realization and lies in the range

$$\lceil -N \log_2 p(\mathbf{A}) \rceil \leq N_c \leq \lceil -N \log_2 p(\mathbf{B}) \rceil + 2$$

where $\mathbf{A}$ is the most probable symbol and $\mathbf{B}$ is the least probable symbol. For example, for the case $p(\mathbf{A}) = 2/3$, $p(\mathbf{B}) = 1/3$, and $N = 16$, the range of possible output codeword length in bits is [10,28]. The upper bound of the range of codeword length, $N_{c,max} = \lceil -N \log_2 p(\mathbf{B}) \rceil + 2$, determines the range and size of key.

The key scheduler utilizes a key sequence of length N_k and generates a sequence with a very long period using repeated XOR operations through

$$b_{N_k+i} = b_i \oplus b_{N_k+i-1}, \qquad \text{for } i = 1, 2, \ldots \tag{1}$$

where b_i is the ith bit of the long-period key sequence and the first N_k bits are from the input key sequence. This can be implemented in software by simple bit operations and in hardware by using a Linear Feedback Shift Register (LFSR), and generates a binary sequence with a very long period of $2^{N_k} - 1$. The long-period key sequence has the same entropy as the N_k bits used to generate it, and in fact any consecutive N_k bits can be used to obtain the corresponding N_k bits of the input key sequence. However, use of the long-period sequence prevents keys used for interval splitting or permutation from having a short period, making it more difficult to mount attacks using periodic input sequences or the periodicity of the permutation key. It should also be noted that while the foregoing discussion is based on permutations using rows of four bits or input symbols, rows of other sizes can be used as well. In general, for a given input size, utilizing a permutation block chosen to have approximately equal row and column dimensions will minimize the permutation key length.

III. SECURITY

Assessing security can be challenging in any encryption system because showing robustness against known attacks does not preclude the existence of unknown attacks against which the system may not be robust. This has been a longstanding issue even with mature encryption standards such as AES, which is currently considered secure but for which future security cannot be guaranteed. As Courtois notes in [12], "Our guess is rather that all the block ciphers with 256-bit keys that were submitted to AES, will some day be broken faster than by exhaustive search, simply because our current knowledge about the real security of block ciphers is yet very low."

In light of the inherent challenge to proving security, we adopt an approach here that, as was done during the development of AES, considers known attacks and ensures that they cannot be used successfully. In the context of a secure arithmetic coder, potential weaknesses lie in the ability to correlate the input symbol stream with attributes of the output binary codeword and to use those correlations to infer key information. The core of the encoder, the Interval Splitting AC, when implemented without any input permutation and codeword permutation, can be attacked using carefully constructed sequences that reveal split locations. When the input permutation is added, the task of the attacker is complicated by the need to first determine the permutation, and then to determine split locations. Information about the input permutation, however, can be determined by comparing the output for a chosen input string and the outputs due to all other input strings that differ in only one symbol. If, by contrast, an input permutation is not used but an output permutation is used, then analysis of the relative ratios of zeros and ones in output sequences associated with particular input sequences can be used to get information regarding the key sequence. Using both an input and output permutation thwarts all of these attacks and gives a system which appears to be secure. The following describes each of these attacks in more detail and motivates the full system shown in Fig. 1.

A. Chosen Plaintext Attacks for Interval Splitting AC

One of the opportunities to attack a standalone interval splitting AC (i.e., with no permutations) lies in the recursive nature of the splitting and encoding. This can be exploited by an attacker controlling the number of symbols and the specific sequence of symbols to intelligently provide input sequences of varying content and length and gradually gain information on the keys used by the system.

To illustrate how an attacker could gain information by trying different input sequences, denote an input symbol sequence by S_k and the binary number obtained upon encoding that sequence using an interval splitting AC, without any permutations, by

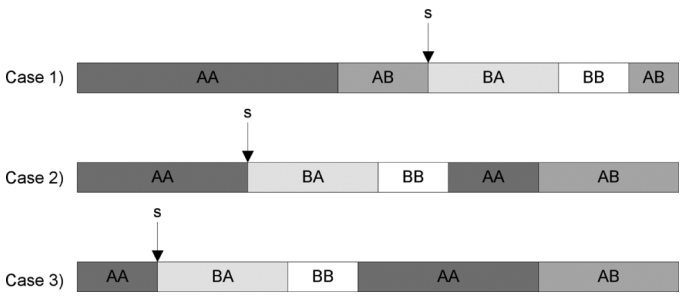


Fig. 5. Possible cases of locations of codewords when $\mathbf{A}$ is split.

$C(S_k)$. Let $I(S_k)$ refer to the intervals for S_k prior to splitting, and $I'(S_k)$ to the intervals for S_k after splitting. For example, in Fig. 2, $I(\mathbf{A})$ indicates the interval marked as $\mathbf{A}$ before splitting as in Fig. 2(a), and $I'(\mathbf{A})$ indicates both of two distinct intervals marked as $\mathbf{A}$ after splitting as in Fig. 2(b).

Consider an interval splitting AC operating on a sequence of N symbols and using splitting key vector $(k_0, k_1, \ldots, k_{N-1})$. For clarity, we assume that when selecting a codeword for an input sequence represented by two disjoint intervals as a result of interval splitting, the encoder will always select a codeword corresponding to a point in the larger of the two intervals. We also assume that interval splitting is not done for the last input symbol in a sequence as there is nothing further to encode.

If symbol $\mathbf{A}$ is split, then after partitioning the two subintervals of symbol $\mathbf{A}$ for $\mathbf{AA}$ and $\mathbf{AB}$, there are three possibilities with respect to the locations of the codewords $C(\mathbf{AA})$ and $C(\mathbf{AB})$, as illustrated in Fig. 5. First, both $C(\mathbf{AA})$ and $C(\mathbf{AB})$ could lie to the left of the contiguous interval for symbol $\mathbf{B}$. This is shown in Fig. 5 as "case 1" and will occur if and only if $\mathbf{A}$ is split at an absolute position s on [0,1) (in contrast with the relative positions used in specifying keys k_i) satisfying $s > p(\mathbf{A}) - (p(\mathbf{AB})/2)$. In other words, the subinterval of $I'(\mathbf{A})$ that is to the right of $I'(\mathbf{B})$ must be shorter than half of $p(\mathbf{AB})$, thereby causing $C(\mathbf{AB})$ to lie to the left of $I'(\mathbf{B})$. Second, $C(\mathbf{AA})$ could be to the left of $\mathbf{B}$ and $C(\mathbf{AB})$ to the right of $\mathbf{B}$ (case 2 in Fig. 5). This will occur if and only if the length of the subinterval of $I'(\mathbf{A})$ that is to the left of $I'(\mathbf{B})$ is larger than half of $p(\mathbf{AA})$, *and* the length of the subinterval of $I'(\mathbf{A})$ that is to the right of $I'(\mathbf{B})$ is larger than half of $p(\mathbf{AB})$. Third and lastly, both $C(\mathbf{AA})$ and $C(\mathbf{AB})$ could lie to the right of $\mathbf{B}$ (case 3 in Fig. 5). This occurs if and only if $\mathbf{A}$ is split at a position s satisfying $s < (p(\mathbf{AA})/2)$, i.e., the subinterval of $I'(\mathbf{A})$ that is to the left of $I'(\mathbf{B})$ must be shorter than half of $p(\mathbf{AA})$.

Clearly, the above analysis also applies with appropriate modifications if symbol $\mathbf{B}$ was split instead of symbol $\mathbf{A}$. When all four possible two-symbol input sequences $\mathbf{AA}$, $\mathbf{AB}$, $\mathbf{BA}$, and $\mathbf{BB}$ are encoded, the following relationships apply.

1) If $C(\mathbf{BX}) < C(\mathbf{AX})$ ('$\mathbf{X}$' can be either $\mathbf{A}$ or $\mathbf{B}$), then either $\mathbf{A}$ was split with $s < (p(\mathbf{AA})/2)$ or $\mathbf{B}$ was split with $s > p(\mathbf{A}) + p(\mathbf{BA}) + (p(\mathbf{BB})/2)$.
2) If $C(\mathbf{AX}) < C(\mathbf{BX})$, then either $\mathbf{A}$ was split with $p(\mathbf{AA}) + (p(\mathbf{AB})/2) \le s < p(\mathbf{A})$ or $\mathbf{B}$ was split with $p(\mathbf{A}) < s < p(\mathbf{A}) + (p(\mathbf{BA})/2)$.
3) If $C(\mathbf{AA}) < C(\mathbf{BX})$ and $C(\mathbf{BX}) < C(\mathbf{AB})$, then $\mathbf{A}$ was split, and $(p(\mathbf{AA})/2) \le s < p(\mathbf{AA}) + (p(\mathbf{AB})/2)$.
4) If $C(\mathbf{BA}) < C(\mathbf{AX})$ and $C(\mathbf{AX}) < C(\mathbf{BB})$, then $\mathbf{B}$ was split, and $p(\mathbf{A}) + (p(\mathbf{BA})/2) \le s < p(\mathbf{A}) + p(\mathbf{BA}) + (p(\mathbf{BB})/2)$.

Thus, by trying only four two-symbol input sequences, the attacker can significantly reduce the possible split locations. By encoding more and longer input sequences, the attacker can gradually narrow the range of possible key sequences and eventually determine the full key.

In the above, it was assumed that an attacker has the flexibility to arbitrarily choose the input sequence length N. However, even if the interval splitting AC is designed to require inputs with minimum size much greater than 2, a generalized version of the above approach can be used. If an attacker can find three input sequences S_1, S_2, and S_3 in which the first symbol of S_1 and S_3 is $\mathbf{A}$ and the first symbol of S_2 is $\mathbf{B}$, and that lead to codewords $C(S_1) < C(S_2) < C(S_3)$, this reveals that interval $\mathbf{A}$ is split when encoding the first symbol. In addition, because the left of the two $\mathbf{A}/\mathbf{B}$ or $\mathbf{B}/\mathbf{A}$ interval boundaries corresponds to the split position as in Fig. 2, and is identified by the key, the split position must be between $C(S_1)$ and $C(S_2)$. By applying more inputs, the possible range of the split position locations can be narrowed until corresponding key for the first symbol can be identified. The split position and corresponding key information for the subsequent symbols can be identified similarly.

Consider the example of an input sequence length $N = 16$ and a key precision of $M = 3$ bits/symbol. As noted earlier, in this case, key locations are equally spaced in each of the eligible splitting intervals. When $M = 3$, key locations k_i are related to three-bit binary keys through $k_i(l) = (2l + 1)/16$ for $l = 0, \ldots, 7$. The relationship between k_0 and an absolute position s in [0,1) is then

$$s = \begin{cases} 2p(\mathbf{A})k_0, & 0 < k_0 < 0.5 \\ p(\mathbf{A}) + 2p(\mathbf{B})(k_0 - 0.5), & 0.5 < k_0 < 1. \end{cases} \quad (2)$$

Table II gives an example of input and output pairs with $N = 16$ and key locations defined as above. Since $C(S_2) < C(S_4) <$

TABLE II
EXAMPLE OF INPUT SEQUENCE AND OUTPUT CODEWORD OF INTERVAL SPLITTING AC

Input sequence	Codeword	Decimal
S_1 : **AAAAAAAAAAAAAAAA**	.1100111001	0.8057
S_2 : **ABAAAAAAAAAAAAAA**	.00010011110	0.0771
S_3 : **ABABAAAAAAAAAAAA**	.0010101000011	0.1644
S_4 : **BAAAAAAAAAAAAAAA**	.01100101010	0.3955
S_5 : **BBAAAAAAAAAAAAAA**	.010010011110	0.2886

TABLE III
AVERAGE NUMBERS OF INPUT SEQUENCE TO FIND WHICH SYMBOL IS SPLIT FOR THE FIRST SYMBOL FOR INTERVAL SPLITTING AC

N	M	Median	Average	Minimum	Maximum
16	3	8	17.6	3	632
16	4	7	30.6	3	2054
16	5	7	56.1	3	8488
16	6	7	100.6	3	19332
8	3	8	15.5	3	256
32	3	7	16.1	3	1571
64	3	7	16.6	3	647

TABLE IV
EXAMPLE OF INPUT SEQUENCES AND OUTPUT CODEWORDS OF INPUT PERMUTATION WITH INTERVAL SPLITTING AC ($N = 16$)

Input sequence	After permutation	Codeword	Decimal $C_d(S_i)$	$\lvert C_d(S_i) - C_d(S_1)\rvert$
S_1: **AAAAAAAAAAAAAAAA**	**AAAAAAAAAAAAAAAA**	.0001111000	0.117188	0
S_2: **ABAAAAAAAAAAAAAA**	**AAAAAAAAAAAAAAAB**	.00011101110	0.116211	0.000977
S_3: **AAAAAAAAAAAABAAA**	**AAAAAAAAAAAAAABA**	.00011101100	0.115234	0.001954
S_4: **AAAAAAABAAAAAAAA**	**AAAAAAAAAAAABAAA**	.00011101010	0.114258	0.00293
S_5: **AAAAAAAAAAAABAAA**	**AAAAAAAAAAABAAAA**	.000111110001	0.121338	0.00415
S_6: **AAAAAAAAAABAAAAA**	**AAAAAAAAAAABAAAA**	.000111001101	0.112549	0.004639
S_7: **AAAAAAAABAAAAAAA**	**AAAAAAAAAABAAAAA**	.001000000001	0.125244	0.008056
S_8: **AAABAAAAAAAAAAAA**	**AAAAAAAAABAAAAAA**	.00110101111	0.105225	0.011963

$C(S_1)$, the fact that an interval corresponding to $\mathbf{A}$ is split is identified. By using the fact $C(S_2) < C(S_3) < C(S_5) < C(S_4)$, the key location for the first symbol must lie between $C(S_3) = 0.1644$ and $C(S_5) = 0.2886$. From (2), possible values of s in interval $\mathbf{A}$ are 0.0833, 0.25, 0.417, and 0.583. Therefore, the key location must be 0.25, $l = 1$ and corresponding key bits can be identified as 001.

Table III presents the results of experiments examining the number of input sequences that need to be tried by an attacker to reveal which symbol is split when encoding the first symbol. The table considers several different combinations of input length N and key precision M. One thousand randomly chosen key sequences were used to obtain the results in each row of the table. Most notably, the median number of trials does not grow with either N or M, though the average and maximum number grow with key precision. This means that some key sequences become stronger against this form of attack as M increases but half of the key sequences are weak regardless of N or M. Furthermore, the use of very large keys would involve overhead that would counter the compression goals of the system.

While the median is an important statistic in examining the vulnerability of randomly selected keys, the information in the table also indicates that careful selection of keys can significantly increase the barriers to an attacker. In particular, while the median values are low and relatively consistent across different values of N and M, the maximum values are many orders of magnitude larger than the median, and also suggest a general correlation between increased N or M and increased robustness. Further examination of the data used to generate the entries in Table III shows that the number of trials needed is highest when the keys are chosen near the edges of the intervals. In addition, greater key precision allows placement of keys closer to the interval edges, thus explaining the growth in maximum values with M. Thus, increased security against an attacker using random key choices would be obtained by using edge-adjacent keys. That said, it should also be noted that a savvy attacker could expect keys to be preferentially located near edges, and could bias a search with that information. The specific advantages that could be gained over a purely random search are of course a function of the search strategy, of which there are an enormous number of possibilities.

As the above discussion illustrates, the primary vulnerability of interval splitting AC alone lies in the ability of an attacker to observe the exact correspondences between the input to the AC and the output bitstreams. In particular, examination of the ordering of codewords provides partial information on the ordering of $I'(S_k)$, as determined by the key. Because attacker can select arbitrary symbol sequences, and because the length of the intervals of $I'(S_k)$ shrinks exponentially as the number of symbols in S_k increases while the possible split positions have a fixed resolution, key information can be revealed.

B. Permutation in Combination With Interval Splitting AC

Permutation offers a simple and powerful way to increase robustness without incurring a compression efficiency penalty. Permutation alone is well known to furnish poor encryption. However, when used in combination with an interval splitting AC, the ability to construct attacks becomes severely compromised. We consider two permutations: the first applied at the input on the string of input symbols and the second applied to the output bitstream.

Consider first an input permutation. An attacker wishing to compromise this system faces an initial challenge to determine the permutation. One way to accomplish this is to first encode a sequence containing only $\mathbf{A}$'s, and then to encode a series of sequences of the same length, but containing exactly one $\mathbf{B}$. Table IV provides an example of such an attack. The first column gives the input sequence and the second column gives the sequence after permutation. The third and fourth columns are the encoder outputs expressed in binary and equivalent decimal form respectively. The final column $\lvert C_d(S_i) - C_d(S_1)\rvert$ gives the distance between the decimal location for the codeword due to the all $\mathbf{A}$ sequence and codewords resulting from inputs containing only one $\mathbf{B}$. The table is sorted in the ascending order according to this distance. In general, the closer the $\mathbf{B}$ lies to the end of the sequence, the smaller its distance will be from the all $\mathbf{A}$'s output. Because of the interval splitting, the correspondence is not strict, as illustrated in this example by the outputs due to S_5 and S_6. However, given enough trials, these general correlations can enable an attacker to infer the permutation used to obtain the second column of the table from the first, and once the permutation is decoded, the interval splitting locations can be determined using the procedures discussed in the previous section.

Now consider a system in which there is no input permutation, but in which an output permutation is applied. Such a system can be attacked by examining information regarding the number of zeros and ones relative to the overall codeword length. As illustrated in Fig. 2, at each encoding step only one symbol is split. In an interval splitting arithmetic coder, one

TABLE V
EXAMPLE OF AVERAGE PERCENTAGES OF MAJORITY BITS FOR THE CASE $N = 16$, $N_a = 8$, AND $N_b = 3$

Symbols	1st search	2nd search	3rd search
AAA	60.0	63.5	66.8
AAB	63.0	64.7	70.3
ABA	**65.5**	57.9	69.5
ABB	60.1	59.3	69.2
BAA	60.6	68.0	75.8
BAB	60.8	**73.3**	73.5
BBA	57.4	63.2	**82.8**
BBB	58.6	64.8	75.3

output codeword will be either the all-zeros or the all-ones sequence. This occurs because when a symbol is split, the two resulting subintervals corresponding to the symbol are pushed to edges of the interval in the previous step. Therefore, the leftmost interval associated with the all-zeros codeword or the rightmost interval associated with the all-ones codeword always correspond to the one possible input sequence (the "split symbol sequence") which consists of all the split symbols. If an attacker succeeds in finding this sequence, then information about split locations is revealed.

An attack utilizes the knowledge that codewords for input sequences which have the same leading symbols as the split symbol sequence start with many zeros or ones. For example, assume that the possible encoding intervals associated with $\mathbf{ABABBBAB}$ are decimal [0,0.0010) and [0.9998,1). Then, before the output permutation, binary codewords corresponding to inputs starting with $\mathbf{ABABBBAB}$ start with at least nine zeros or 12 ones. Output permutation will move these zeros or ones to other locations in the output codeword, but their number will remain fixed. Thus, an attack can use the following steps.

1) Choose plaintexts which vary in the first N_a symbols and are fixed as $\mathbf{A}$ for the remaining $N - N_a$ symbols. Obtain codewords corresponding to the 2^{N_a} plaintexts.
2) Calculate the percentage of majority bits in each codeword. For example, 70% of the bits in the sequence 1100111101 are 1. Obtain an average of the majority bit percentages over input sequences having the same first N_b, where $N_b < N_a$ and can be determined by trial and error.
3) Determine the first N_b symbols of the input sequence by selecting N_b symbols having the maximum average.

Table V gives an example of average percentages of majority bits for the case $N = 16$, $N_a = 8$, and $N_b = 3$. For the first search, $2^{N_a} = 256$ inputs from $\mathbf{AAAAAAAA..A}$ to $\mathbf{BBBBBBBA..A}$ are encoded. Then, data from the 32 codewords whose inputs have the same three starting symbols, but differ in the subsequent five symbols, are averaged. For instance, 65.5%, in the first column of the row labeled $\mathbf{ABA}$ in the table, is obtained by averaging the percentages of majority bits over codewords corresponding to $\mathbf{ABAAAAAAA..A}$ – $\mathbf{ABABBBBBA..A}$. Since this is the maximum among all entries in the first column, in the second searching pass, the first three symbols are fixed at $\mathbf{ABA}$, and the process is repeated, this time with the N_a window lying from the fourth to eleventh symbols and the N_b window comprising the fourth, fifth, and sixth symbols. This gives a maximum as shown in the table for sequence $\mathbf{BAB}$. The window is then moved three symbols to the right, and a third search is performed, giving a maximum at $\mathbf{BBA}$. Therefore, plaintexts starting with $\mathbf{ABABABBBA}$ are chosen for the next step. Since the first nine symbols are fixed and the total sequence length is 16, only $2^7 = 128$ possible symbol sequences remain, and one of these will generate the all-zeros or all-ones codeword. In this example, the input corresponding to the all-zeros codeword, $\mathbf{ABABABBBABBABBAA}$, is identifiable after encoding 892 plaintexts (input sequences). In general, the number of plaintexts needed is approximately $(((N - N_a)/N_b) + 1)2^{N_a}$. Both the search complexity and the reliability of the result increase with N_a.

The input leading to the all-zeros or all-ones sequence is the split symbol sequence, and thus contains information about the interval splitting key. From this attack, it is very hard to directly obtain the codeword permutation key, which is generated from an input key sequence by XOR operations as in (1). However, because split location keys are also generated from the input key sequence, the split symbol sequence is related to XOR results of several bits of input key sequence. Therefore, an attacker can use this information to reduce the number of possible input key sequences that must be tried.

Utilizing both an input and an output permutation gives a system that defeats both of the attacks usable on systems with only one permutation, and appears to be robust against chosen plaintext attacks. The attacks available on an interval splitting AC alone and on a system with input permutation followed by interval splitting AC cannot be applied because codeword comparisons are meaningless due to codeword permutation. The attacks used for codeword permutation cannot be applied because an attacker needs to find leading symbols first, but symbol order is scrambled by the input permutation.

IV. IMPLEMENTATION COMPLEXITY

The implementation complexity of the permutation-based system in Fig. 1 is dominated by the interval splitting arithmetic coder because permutations and key scheduling utilize very basic bit operations. Permutations are implemented through circular shifts which are easily implemented in either hardware or software. The key scheduler employs XOR operations and can operate in one of two modes. First, a key sequence can be generated once before encoding begins and used for multiple input blocks until a new input key sequence is sent to a decoder. Alternatively, different keys can be generated for each input block. This second approach involves slightly more computing power and increases the burden on a potential attacker.

An interval splitting AC can be implemented utilizing techniques similar to those used in traditional arithmetic coding and can benefit from the same optimizations for speed, finite precision, etc. Examples of such techniques include simple table lookups used for performing the computationally critical operations of interval subdivision and probability estimation [13], mechanisms that improve throughput by allowing renormalization to occur during the arithmetic operations [14], and encoding of two or more symbols in a single cycle [15]. All these optimizations can also be applied to an interval splitting

AC. The main difference lies in the number of intervals, which approximately doubles (the precise number is one fewer than twice the original number, because the interval at the center of the [0,1) range is not split), which doubles the amount of memory needed to store interval start and end locations. In addition, renormalization must utilize both the interval length and the key, introducing an additional multiplication, though as with traditional arithmetic coding, faster algorithms that replace the multiplications with simpler operations can be introduced [16], [17].

At some cost in security, the complexity could be reduced by replacing the interval splitting AC in Fig. 1 with the RAC coder in [7]and [8]and leaving all other permutation steps in place as in the figure. The implementation complexity of RAC is only marginally higher than traditional arithmetic coding, since the main added task of tracking interval swaps is very simple, and the number of intervals is not increased. However, the security of RAC is lower because the key sequence controls only the ordering of intervals while in interval splitting AC it controls the ordering and the length of intervals. For example, if an input corresponding to an all-zero sequence is revealed in RAC, an attacker then knows all information of intervals. For the same condition in interval splitting AC, the attacker still needs to identify split locations. Moreover, RAC becomes simple XOR when $p(\mathbf{A}) = p(\mathbf{B}) = 1/2$. Hence, if an attacker has the ability to control the input symbol probabilities assumed by the RAC, the attacker can easily obtain permutation information. As noted earlier, however, the RAC was never designed for security, so it is unsurprising that it is less robust than the interval splitting AC, which was designed with security as a specific goal.

Given that the goals of the system described here are both security and compression, it is relevant to consider a system consisting of a traditional arithmetic encoder followed by AES, which of course would also deliver security and compression. Since AES was designed for efficient hardware implementation, it is extremely fast when it is fully pipelined in hardware [18]. However, because a traditional arithmetic coder needs to work sequentially—i.e., the results of encoding earlier symbols are necessary before encoding of later symbols can be performed—the AC cannot easily be parallelized and becomes the bottleneck in a combined AC/AES system. In addition, number of transformation steps is significantly higher in AES than in the permutations of the system of Fig. 1. AES consists of 40 sequential transformation steps composed of simple and basic operations such as table lookups, shifts, and XORs. For a block size of $N = 128$, these steps require a total 19 shifts (assuming parallel operations when possible), use of 336 bytes of memory, and the XORing of approximately (the exact requirement is data dependent) 608 bytes of data. In contrast, for an interval splitting arithmetic coder operating on an input of size 128, the permutations in Fig. 1 require six shifts (again assuming parallel operation when possible), a maximum of 40 bytes of memory, and no XOR operations. Moreover, while interval splitting AC has more complexity than traditional AC as described previously, calculations of two subintervals can be performed in parallel and each calculation is similar to its equivalent in a traditional arithmetic coder. Thus, in terms of throughput the system in Fig. 1 can be faster than traditional AC followed by AES because in Fig. 1 the sequential steps are fewer and are performed during the arithmetic coding.

TABLE VI
COMPARISON OF CODE LENGTHS AS A FUNCTION OF SEQUENCE LENGTH N

Symbol Prob.	N	$H \times N$	Traditional	This paper	Efficiency penalty in %
$p(\mathbf{A})$=2/3	10	9.183	10.21	10.87±.06	6.38±.54
H=Entropy=.9183	1000	918.296	919.10	919.71±.08	0.07±.01
$p(\mathbf{A})$=9/10	10	4.690	5.58	6.58±.01	17.91±.18
H=Entropy=.4690	1000	468.996	470.15	470.64±.01	0.10±.00

Another advantage lies in the flexibility with respect to codeword lengths. Although the average length of codewords is easily determined from the entropy, in practice any given input sequence realization will not typically possess precisely "average" characteristics, and in many environments the probability model itself is wrong. So the assumed probabilities internal to an AC differ from those actually provided as inputs even when many input symbols are encoded. This could be problematic when AC is followed by AES because the block nature of AES could require zero padding and an accompanying loss of overall efficiency. This problem could be circumvented by replacing AES with a stream cipher, which by definition places no constraints on sequence length. However, stream ciphers with comparable complexity to AES are not as strong as AES, meaning that the elimination of the block processing requirement would come at some cost to overall security.

V. EXPERIMENTAL RESULTS

The permutation steps involve no cost to coding efficiency, so the only efficiency differences with respect to traditional arithmetic coding arise due to the interval splitting, which as discussed earlier, results in slightly increased bounds on codeword length. Table VI shows the results of encoding input sequences with length $N = 10$ and 1000 symbols and allows an efficiency comparison in absolute and relative terms with traditional arithmetic coding. The upper half of the table considers the case where $p(\mathbf{A}) = 2/3$, and the lower half of the table considers the case where $p(\mathbf{A}) = 9/10$. The code lengths shown in the table are averages based on simulations using 1000 random sequence realizations. Since the exact length of the output when interval splitting is used depends not only on the input data but also on the specific sequence of split locations used, the column labeled "Interval Splitting" gives the mean and standard deviation of the code lengths based on randomized key locations for splitting. The rightmost column in the table gives the corresponding mean and standard deviation for the efficiency penalties. As expected, the penalty and the variance of the penalty quickly become very small as sequence length increases.

Fig. 6 presents results in graphical form for input sequences with $p(\mathbf{A}) = 2/3, 6/7$, and $9/10$. As with the data in Table VI, the curves in the figure are averages based on simulations using 1000 random sequence realizations. The results confirm that the efficiency penalty in percentage terms becomes smaller with increasing N, falling to approximately 0.6% for $N = 100$ and to 0.007% for $N = 10000$ when $p(\mathbf{A}) = 2/3$. In absolute

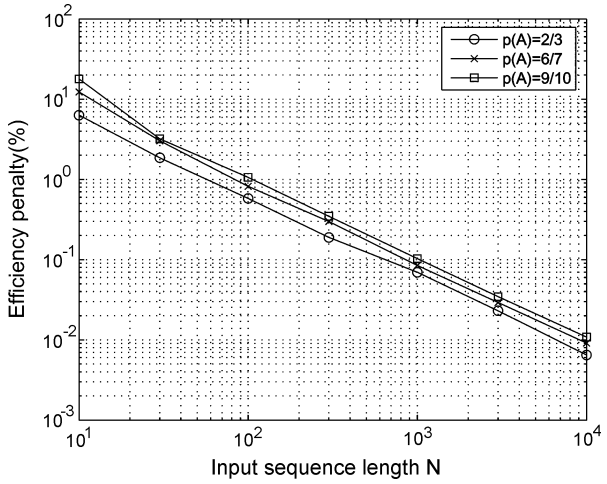


Fig. 6. Efficiency penalty of interval splitting method as a function of sequence length N.

terms, the efficiency penalty with respect to traditional arithmetic coding is well under one bit, and typically closer to 0.5 bits (for the entire sequence of N symbols). In addition, it should also be noted that the modest efficiency penalty is the cost of obtaining secrecy. It arises because of the disjoint nature of the intervals, which in turn contributes to the secrecy since it introduces a perturbation on the traditionally more direct association between symbol string probability and the length of the resulting representation.

Interesting areas of possible extension of this work include generalization to M-ary alphabets, use of adaptive probability tables and context-based coding. For the special case of first-order binary coding, the coder presented here can be used if the input sequence is first differentially processed to create a sequence that expresses data changes (e.g., 0 if the current bit is the same as the previous bit, 1 if the current bit differs from the previous bit). While this causes some efficiency loss to the extent that the transition probabilities may be asymmetric, in practice the method can work quite well. As an example, we performed compression using interval splitting AC on the "pic" binary image file available in the Calgary Corpus [19]. Zeros in this image occur with probability 0.923, corresponding to a zero-order "entropy" rate of 0.3915 bits per image bit. When coded using a transition-based interval splitting AC, however, the coding achieved is a rate of 0.1546 bits per image bit (in other words, a compression ratio of 6.47 : 1), which is within 1.4% of the true first-order entropy, and within 1.4% of the result noted in [19].

More generally, utilizing full-context information would require modifications to the encoder and reconsideration of the input permutation as that permutation can destroy the very dependencies that are the basis for contextual knowledge. One option would be to process the data in blocks sufficiently large to preserve context information. Of course, regardless of the size of the block, the first symbol in the block will be stripped of any context information. Thus, the tradeoff of block-based input permutation lies between security, which is weakened when larger blocks are used, and efficiency, which benefits from larger blocks because more of the symbols in the block have full-context information. In the M-ary case it may also be possible to simply dispense with the input permutation when M is reasonably large. This is because when there are many more intervals, the number of potential split locations is also larger. For the binary case with an input sequence of length 10, an attacker can easily examine all possible input sequences, something which is clearly impossible for a sequence of the same length when the alphabet size is $M = 256$. Thus, the sheer number of input possibilities adds to the security, and would potentially enable the overall system to remain secure in the presence of an output permutation alone.

VI. Conclusion

An arithmetic coder in which the intervals associated with each symbol combination are split in accordance with a key, and in which permutations are applied both to input symbol sequence and to the output binary sequence, has been presented. The system offers both compression and security, and thwarts all known attacks aimed at obtaining information about the input or output permutation or the interval splitting keys. For each encoded symbol, a pair of intervals is split, and this split can occur in parallel. So the throughput can be identical to that of a traditional arithmetic coder. The permutations add negligible complexity. The code length increase induced by interval splitting is bounded to less than one bit per N-symbol sequence, and in practice the increase is often approximately 0.5 bits per N-symbol sequence. In percentage terms this efficiency penalty becomes vanishingly small as N increases. Thus, for sequences of reasonable length, the efficiency cost for obtaining security is negligible. While we have focused on the static binary case for simplicity, the methods presented here can also be applied to M-ary and/or adaptive arithmetic coding.

References

[1] G. Langdon and J. Rissanen, "Compression of black-white images with arithmetic coding," *IEEE Trans. Commun.*, vol. COM-29, no. 6, pp. 858–867, Jun. 1981.

[2] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice*. Norwell, MA: Kluwer Academic, 2002.

[3] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[4] J. Cleary, S. Irvine, and I. Rinsma-Melchert, "On the insecurity of arithmetic coding," *Comput. Secur.*, vol. 14, no. 2, pp. 167–180, 1995.

[5] H. Bergen and J. Hogan, "A chosen plaintext attack on an adaptive arithmetic coding algorithm," *Comput. Secur.*, vol. 12, no. 2, pp. 157–167, Mar. 1993.

[6] X. Liu, P. Farrell, and C. Boyd, "Unified code," in *Proc. Int. Conf. Cryptography Coding*. Berlin, Germany: Springer-Verlag, 1999, vol. 1746, Lecture Notes in Computer Science, pp. 84–93.

[7] M. Grangetto, A. Grosso, and E. Magli, "Selective encryption of JPEG2000 images by means of randomized arithmetic coding," in *Proc. IEEE 6th Workshop on Multimedia Signal Processing*, Siena, Italy, Sep. 2004, pp. 347–350.

[8] M. Grangetto, E. Magli, and G. Olmo, "Multimedia selective encryption by means of randomized arithmetic coding," *IEEE Trans. Multimedia*, vol. 8, no. 5, pp. 905–917, Oct. 2006.

[9] J. Wen, H. Kim, and J. D. Villasenor, "Binary arithmetic coding with key-based interval splitting," *IEEE Signal Process. Lett.*, vol. 13, no. 2, pp. 69–72, Feb. 2006.

[10] *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, Fed. Inf. Process. Standards Pub. 197, 26, NIST, Nov. 2001.

[11] T. Cover and J. Thomas, *Elements of Information Theory*. New York: Wiley, 1991.

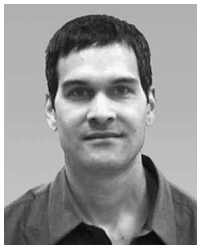
[12] N. T. Courtois, "General principles of algebraic attacks and new design criteria for cipher components," in *Proc. AES 2004*. Berlin, Germany: Springer-Verlag, 2005, vol. 3373, Lecture Notes in Computer Science, pp. 67–83.
[13] D. Marpe and T. Wiegand, "A highly efficient multiplication-free binary arithmetic coder and its application in video coding," in *Proc. ICIP 2003*, Barcelona, Spain, Sep. 2003, pp. II-263–II-266.
[14] M. Dyer, D. Taubman, and S. Nooshabadi, "Improved throughput arithmetic coder for JPEG2000," in *Proc. Int. Conf. Image Process.*, Singapore, Oct. 2004, pp. 2817–2820.
[15] R. R. Osorio and J. D. Bruguera, "A new architecture for fast arithmetic coding in H.264 advanced video coder," in *Proc. 8th Euromicro Conf. Digital System Design*, Porto, Portugal, Aug. 2005, pp. 298–305.
[16] J. Rissanen and K. M. Mohiuddin, "A multiplication-free multialphabet arithmetic code," *IEEE Trans. Commun.*, vol. 37, no. 2, pp. 93–98, Feb. 1989.
[17] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Trans. Inf. Sys.*, vol. 16, no. 3, pp. 256–294, Jul. 1998.
[18] A. Hodjat and I. Verbauwhede, "Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors," *IEEE Trans. Comput.*, vol. 55, no. 4, pp. 366–372, Apr. 2006.
[19] M. Powell, 2006, The Canterbury Corpus [Online]. Available: http://corpus.canterbury.ac.nz/index.html

Hyungjin Kim (S'06) received the B.S. and M.S. degrees in electrical engineering from the Seoul National University, Seoul, Korea, in 1997 and 1999, respectively. He is currently pursuing the Ph.D. degree at the University of California, Los Angeles.

His research interests include compression and encryption algorithms, communications, and video image processing.

Jiangtao Wen (S'94–A'96–M'03–SM'05) received the B.S., M.S., and Ph.D. degrees with honors from Tsinghua University, Beijing, China, all in electrical engineering.

From 1996 to 1998, he worked as a Staff Research Associate at the University of California, Los Angeles, where he conducted cutting-edge research in communications and image and video coding. He currently works for Ortiva Wireless Inc., La Jolla, CA, where he is leading the development of world-class wireless multimedia delivery technologies. Prior to Ortiva, he was the Director of Video Codec Technology of Mobilygen Corp., a venture-funded fabless semiconductor company based in Santa Clara, CA, where he played an instrumental role in the development of low-power, high-quality, standard-compliant H.264 solutions. Earlier in his career, he worked as the Principal Scientist of PacketVideo Corp., and managed PacketVideo's algorithm development and international standards efforts. He is an expert in multimedia communication, DRM, image and video coding and communications. He has authored many widely referenced papers and holds nine granted U.S. patents with numerous others pending.



John D. Villasenor (S'89–M'89–SM'97) received the B.S. degree from the University of Virginia, Charlottesville, in 1985 and the M.S. and Ph.D. degrees from Stanford University, Standford, CA, in 1986 and 1989, all in electrical engineering.

From 1990 to 1992, he was with the Radar Science and Engineering Section of the Jet Propulsion Laboratory, Pasadena, CA, where he developed methods for imaging the Earth from space. He joined the Electrical Engineering Department, University of California, Los Angeles (UCLA) in 1992, and is currently a Professor. He served as Vice Chair of the Electrical Engineering Department from 1996 to 2002. At UCLA, his research efforts were in communications, computing, imaging and video compression, and networking.