mm18 manuscript JPEG Decompression in the Homomorphic Encryption Domain

Xiaojing Ma
Service Computing Technology and System Lab,
Cluster and Grid Computing Lab,
School of Computer Science and Technology,
Huazhong University of Science and Technology
Wuhan, China
lindahust@hust.edu.cn

Changming Liu
Service Computing Technology and System Lab,
Cluster and Grid Computing Lab,
School of Computer Science and Technology,
Huazhong University of Science and Technology
Wuhan, China
webmaster@marysville-ohio.com

Ben Zhu*
Microsoft Research Asia
Beijing, China
binzhu@microsoft.com

Sixing Cao
Service Computing Technology and System Lab,
Cluster and Grid Computing Lab,
School of Computer Science and Technology,
Huazhong University of Science and Technology
Wuhan, China
lleipuner@researchlabs.org

ABSTRACT

Privacy-preserving processing is desirable for cloud computing to relieve users' concern of loss of control of their uploaded data. This may be fulfilled with homomorphic encryption. With widely used JPEG, it is desirable to enable JPEG decompression in the encryption domain. This is hard to achieve since JPEG decoding needs to determine a matched codeword, which then determines a codeword-dependent number of coefficients. With no access to encrypted content, a decoder does not know which codeword is matched, and thus cannot tell how many coefficients to extract, not to mention to compute their values. In this paper, we propose a novel scheme that enables JPEG decompression in the homomorphic encryption domain. The scheme applies a statically controlled iterative procedure to decode one coefficient per iteration. In an iteration, each codeword is compared with the bitstream to compute an encrypted boolean that represents if the codeword is a match or not. only one codeword's boolean is an encrypted one. Each codeword would produce an output coefficient and generate a new bitstream by dropping consumed bits as if it were a match. If a codeword is associated with more than one coefficient, the codeword is replaced with another codeword representing the remaining undecoded coefficients for the next decoding iteration. The summation of each codeword's output multiplied by its matching boolean is the output of the current iteration. This is equivalent to selecting the output of the matched codeword. A side benefit of our statically controlled decoding procedure is that paralleled Single-Instruction Multiple-Data (SIMD) is fully supported, wherein multiple plaintexts are encrypted into a single plaintext, and decoding a ciphertext block corresponds to decoding all corresponding plaintext blocks. SIMD reduces the total size of ciphertexts of an image. Experimental results are reported to show the performance of our proposed scheme.

*This author is the one corresponding author.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
MM'18, October 2018, Seoul, Korea
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**;

KEYWORDS

Privacy-preserving decompression, Homomorphic encryption, JPEG decoding.

ACM Reference Format:
Xiaojing Ma, Changming Liu, Ben Zhu, and Sixing Cao. 2018. mm18 manuscript JPEG Decompression in the Homomorphic Encryption Domain. In *Proceedings of ACM Multimedia (MM'18).* ACM, Seoul, Korea, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the era of cloud computing, a growing amount of data is sent to public clouds for storage and processing. Public clouds are typically operated by third parties. Loss of control of the uploaded data is a critical concern for many current and potential cloud users. To address this issue, privacy-preserving cloud processing has been actively studied in recent years. Among the proposed approaches, Homomorphic Encryption (HE) is a promising approach. It allows computation on ciphertexts, with the result, when decrypted, matching the result of the operations on the plaintexts [14]. By processing ciphertexts directly, a desired result is obtained while privacy is fully preserved.

Many useful algorithms have already been enabled to execute on ciphertexts of homomorphic encryption, such as some simple statistical functions (e.g., sum, sum of squares, logistical regressions)

[20], AES [15], linear transforms (e.g., discrete cosine transform [6], discrete Fourier transform [5]), and bubble sort [2]. These enabled algorithms require linear operations and a bounded number of multiplications on ciphertexts. However, many more operations and algorithms have yet be enabled on ciphertexts, such as division, square roots, etc. Run-length encoding using fully homomorphic encryption has been recently studied [9]. It concludes that running run-length encoding (or any lossless compression algorithm) in the homomorphic encryption domain would not compress (i.e., achieve the worst-case compression ratio) unless some information is lost.

Decompression in the homomorphic encryption domain has yet been studied. Studying decompression of some practical multimedia compression in the homomorphic encryption domain is valuable not only theoretically but also practically since, if enabled, it would allow the following privacy-preserving cloud processing: Multimedia data is typically compressed in real world applications due to a large amount of data. To protect privacy, compressed multimedia data should be encrypted before sending to a cloud. The cloud should decompress the uploaded compressed data in the encryption domain before applying other privacy-preserving operations on the encrypted multimedia data.

It is a great challenge to enable decompression in the encryption domain. A JPEG decoder, for example, compares a bitstream with a Huffman table of variable-length codewords to find a matched codeword, which in turn determines a codeword-dependent number of Discrete Cosine Transform (DCT) coefficients and their values and consumes a codeword-dependent number of coding bits. This process is applied iteratively until 64 (or a desirable number of) DCT coefficients of each block in an image have been determined. With no access to any encrypted content, a JPEG decoder does not know which codeword is actually matched, and thus cannot determine how many DCT coefficients it should extract in an iteration, not to mention to determine their values or to know when to stop decoding for the current block and move to decoding for the next block. Prior art offers no clue on how to address this challenge.

In this paper, we propose a novel scheme that enables JPEG decompression in the encryption domain of leveled or fully homomorphic encryption. We devise a statically controlled iterative decoding procedure that decodes one coefficient per iteration using only sequential operations. Dynamically controlled operations used in a conventional JPEG decoder are completely removed. In one iteration, each codeword is compared with the bitstream to compute an encrypted boolean that represents if the codeword is a match or not. Among all the codewords in a Huffman table, there is one and only one codeword that is a match, i.e., its boolean is an encrypted one. Each codeword would process as if it were a match, such as producing an output coefficient and generating a new bitstream by dropping consumed coding bits for the next iteration. If a codeword is associated with more than one coefficient, the codeword is replaced with another codeword that represents the remaining undecoded coefficients in generating a new bitstream for the next decoding iteration. To fulfill this operation, additional codewords may be added temporally to a Huffman table during decoding. At the end of the iteration, we take summation of each codeword's output, such as its output coefficient or new bitstream, multiplied by its matching boolean as the output of the iteration. This summation is equivalent to selecting the output of the matched codeword.

A side benefit of our statically controlled decoding scheme is that paralleled decoding of Single-Instruction Multiple-Data (SIMD) is fully supported: multiple, such as $D = 256$, plaintexts are encrypted into a single ciphertext, and decoding one ciphertext block corresponds to decoding simultaneously all the corresponding plaintext blocks. This would also reduce the number of ciphertexts by D times for an encrypted JPEG image.

Without loss of generality, we focus on the baseline JPEG in this paper, wherein an image is of 8 bits per color components (i.e., 8-bit images).

This paper is organized as follows: we first introduce homomorphic encryption and JPEG in Section 2, and then present our Huffman decoding process in Section 3 and remaining JPEG decoding operations in Section 4. SIMD and encryption scheme are described in Section 5. Implementation and experimental results are reported in Section 6, and the paper concludes in Section 7.

2 BACKGROUND

2.1 Homomorphic Encryption

Partially homomorphic encryption supports homomorphic additions or multiplications, but not both. Paillier [21] supports only homomorphic additions, i.e., linear operations. It has been widely used in early privacy-preserving schemes [3–6, 10, 16, 19, 22, 26], which would resort to multi-party computation (MPC) [12] for unsupported non-linear operations. MPC requires involving the party holding the private key during computation.

Fully Homomorphic Encryption (FHE) was first constructed by Gentry [14] to support an arbitrary number of homomorphic additions and multiplications, but a costly procedure called bootstrapping is needed. BGV [7] is a Leveled Homomorphic Encryption (LHE) scheme that supports homomorphic additions and multiplications of polynomials with a bounded degree on Z_n without resorting to bootstrapping. On Z_2, addition and multiplication are equivalent to binary XOR and AND operations, respectively. This paper focuses on enabling the JPEG decompression with homomorphic additions and multiplications of a polynomial with a bounded degree on Z_2.

With XOR and AND gates, an FHE computer can perform many common operations on n-bit integers, such as addition, subtraction, multiplication, sign change, left and right shift (with or without sign extension), comparisons, and so on [13]. These operations can be realized with polynomials of a bounded degree, and thus can run with LHE. On the other hand, common operations and algorithms, such as division of two real numbers, trigonometric and exponential functions, comparison of two real numbers, cannot be implemented in polynomials of a bounded degree. They cannot run with LHE. They are either approximated with polynomial algorithms or resorted to MPC when running with LHE.

Unlike a normal computer, the plaintext of an evaluation result remains inaccessible to an FHE computer. This subtle difference makes the FHE unable to execute any content-based operations, such as a dynamic control flow (e.g., "if ... then ... else" statement), with a condition depending on an encrypted value. In

general, an FHE computer can only execute programs of static control structures (i.e., programs that can be converted into a linear sequence of instructions). There are a number of regularization techniques to convert a dynamic control structure into a static one. For example, the following conditional assignment operation, $x = c\ ?\ a : b$, can be converted to an equivalent static structure: $x = (c\ AND\ a)\ XOR\ ((1\ XOR\ c)\ AND\ b)$ [2].

Single-Instruction Multiple-Data (SIMD)[23], supported by BGV, is a technique to pack multiple plaintexts into one ciphertext, which significantly reduces the size of ciphertexts and allows parallelized processing. Apparently, parallelized processing in SIMD requires that operations on each packed plaintext are identical.

For more information on homomorphic encryption, interested readers are referred to a recent review paper [1].

2.2 JPEG

JPEG [24] is a standard image compression scheme that has been widely used in practice. At encoding, an image is partitioned into 8×8 blocks. Each block is DCT-transformed and quantized. The 64 quantized DCT coefficients in each block are then ordered zig-zag wise, transformed into a sequence of intermediate symbols before entropy and variable-length encoded. For AC coefficients, a pair of symbols are used to represent a nonzero AC coefficient and its preceding zero-valued AC coefficients:

$$S_1 : (Runlength, Size) \qquad S_2 : (Amplitude)$$

where $Runlength \in [0, 15]$ represents the zero-run, i.e., the number of consecutive zero-valued AC coefficients before the nonzero AC coefficient, $Size$ is the number of bits taken by $Amplitude$, and $Amplitude$ represents the value of the non-zero AC coefficient. When an actual zero-run is over 15, $S_1 = (15, 0)$ is used to represent a zero-run of 16. A special symbol, $(0, 0)$, is used to indicate end of block (i.e., the remaining AC coefficients are all zeros). This symbol is thus referred to as the EOB (End Of Block) symbol.

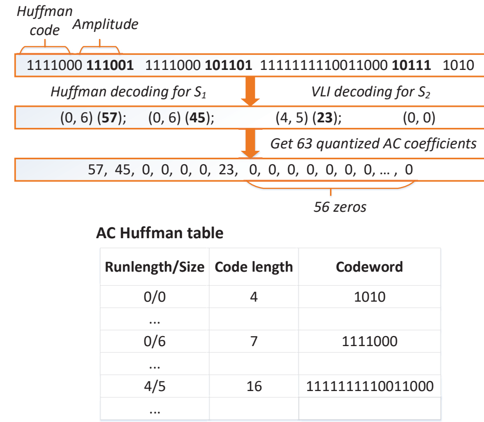
DC coefficients are differentially encoded, with each differential DC coefficient represented by the two symbols:

$$S_1 : (Size) \qquad S_2 : (Amplitude)$$

where $Size$ and $Amplitude$ are defined similarly as their counterparts for AC coefficients.

For both AC and DC coefficients, S_1 is Huffman-encoded, while S_2 is Variable-Length Integer (VLI) encoded. Both encodings result in variable-length codewords.

JPEG decompression reverses the aforementioned encoding process to recover the image from a bitstream of JPEG-compressed image. To recover 63 quantized AC coefficients in a block, a bitstream is compared with an AC Huffman table to determine a matched codeword, which in turn determines $Runlength$ and $Size$ in S_1. $Runlength$ tells the number of preceding zero-valued AC coefficients, and $Size$ tells the number of bits to read for $Amplitude$ to determine a quantized nonzero AC coefficient. The differential DC coefficient in a block is recovered in a similar manner. The quantized DC coefficient in the block is recovered by adding the differential DC coefficient to the quantized DC coefficient of the preceding block. The recovered quantized DCT coefficients in each block are then dequantized, transformed with an inverse DCT, and rounded to the valid range of pixel values.



Runlength/Size	Code length	Codeword
0/0	4	1010
...		
0/6	7	1111000
...		
4/5	16	1111111110011000
...		

Figure 1: An example of decoding 63 quantized AC coefficients of a 8×8 block.

Fig.1 shows the process to decode AC coefficients of a block. In comparing the bitstream with codes in the AC Huffman table, $S_1 = (0, 6)$ with $codeword = 1111000$ is matched. It means there is no preceding zero and the number of bits for S_2 is 6. By VLI-decoding the subsequent 6 bits, $S_2 = (57)$ is obtained. By applying this procedure iteratively, another two pairs of symbols, $S_1 = (0, 6), S_2 = (45)$ and $S_1 = (4, 5), S_2 = (23)$ are recovered, and then $(0, 0)$, i.e., EOB, is matched, which indicates the remaining AC coefficients are all 0. These symbols are then converted to 63 quantized AC coefficients of the block: $57, 45, 0, 0, 0, 0, 23, 0, \ldots, 0$.

3 HUFFMAN DECODING IN HOMOMORPHIC ENCRYPTION DOMAIN

3.1 Challenges and Our Solution

From the decoding procedure described in Section 2.2, there is a dynamic control structure in each iteration to decode a set of symbols S_1 and S_2, wherein a different number of bits in a bitstream is consumed and a different number of quantized DCT coefficients is recovered depending on the matched Huffman codeword. As mentioned in Section 2.1, a FHE computer does not support dynamic control structures, and there is a subtle difference between a FHE computer and a normal computer. Although comparison of a bitstream with each codeword in a Huffman table can be executed by a FHE computer, the inaccessibility of which codeword is actually matched makes a FHE computer unable to determine the number of recovered DCT coefficients and their values recovered as well as the number of bits consumed in an iteration, or to determine when one block's decoding is done and the next block's decoding should start. This means that a FHE computer cannot execute the JPEG decoding procedure described in Section 2.2 even when the known regularization techniques are applied to convert dynamic control structures to static ones.

By examining the bottleneck for enabling a FHE computer to execute the JPEG decoding process, we came up with a novel approach of regularizing the JPEG decoding: instead of producing a variable number of DCT coefficients per iteration, we would like to recover one quantized DCT coefficient per iteration, resulting in a

fixed number of iterations per block, i.e., 63 iterations for AC coefficients and one iteration for DC coefficient. To achieve this goal, a matched codeword associated with $n(> 1)$ coefficients is replaced by a codeword associated with the remaining $n - 1$ coefficients in each decoding iteration. For example, if the codeword of $S_1 = (3, 4)$ is matched in an iteration, the codeword is then replaced by the codeword associated with $S_1 = (2, 4)$, and a zero-valued coefficient is output for the current iteration. This process may require adding additional codewords to a Huffman table for proper decoding. For example, when the codeword of $S_1 = (15, 0)$, which means 16 consecutive zeros, is matched, the codeword should be replaced with a codeword representing 15 consecutive zeros[1]. Such a codeword does not exist in a Huffman table. Additional codewords can be added in several ways, as described in Section 3.3.

With the above regularization technique, conditional branch operations in a conventional JPEG decoding procedure, with different branches consuming a different number of bits and recovering a different number of quantized DCT coefficients, are regularized to a static control structure program. The identical sequential operations in an iteration brings on a side benefit: our JPEG decoding scheme enables parallelized decoding in SIMD, wherein decoding one DCT coefficient on ciphertexts corresponds to decoding multiple plaintext coefficients. This parallelized decoding is unimaginable for a conventional JPEG decoding procedure.

3.2 Regularized Huffman Decoding

The encryption scheme will be described in Section 5. For the time being, we assume that each bit in a payload of a JPEG bitstream is individually encrypted with a homomorphic encryption scheme such as BGV, while Huffman tables are not encrypted. For an encrypted bitstream of an 8×8 block, the operations shown in Fig. 2 are applied to derive the 64 quantized DCT coefficients of the block, which are still encrypted. At the beginning of each iteration to decode one DCT coefficient in a block, it checks if it has already output 64 quantized DCT coefficients. If the answer is positive, decoding the current block is done. Otherwise it calculates an encrypted bit $[b]$ for each codeword in the current Huffman table, which indicates if the codeword matches the current bitstream or not (see Section 3.2.1). Then it calculates a coefficient as the output of the current iteration (see Section 3.2.2), and generates a new bitstream for the next iteration unless it is the last iteration for the block (see Section 3.2.3). Instead of using an encrypted counter to indicate the starting position of the bitstream for the current iteration like in a conventional JPEG decoding procedure, we adopt an approach to drop consumed bits, with a new bitstream generated at the end of each iteration to serve as the bitstream for the next iteration. In this way, the bitstream at each iteration always starts from the very beginning. This approach simplifies the decoding scheme.

In the following description, Z_2 is used: each plaintext is a bit, with addition (+) and multiplication ($*$) being the operations on Z_2, i.e., *XOR* gate and *AND* gate, respectively. We use a pair of square brackets, "[ ]", to represent ciphertext and a subindex (staring from 0) to indicate the position of an element in a collection. For example, $[b]$ means a ciphertext of plaintext b, and $[bits]_0$ means the first encrypted bit in an encrypted bitstream $[bits]$. For simplicity, a collection of plaintexts inside a pair of square brackets denotes a collection of their corresponding ciphertexts, and an integer inside a pair of square brackets denotes binary encryption of the integer's bits in two's complement representation . For example, [010110] denotes a sequence of ciphertexts [0][1][0][1][1][0], and [22] $\equiv$ [010110] $\equiv$ [0][1][0][1][1][0] if we use 6 bits to represent an integer.

[1]This codeword differs from a codeword of (15, k) with $k > 0$ since the subsequent coefficient of (15, 0) may not be a nonzero coefficient.

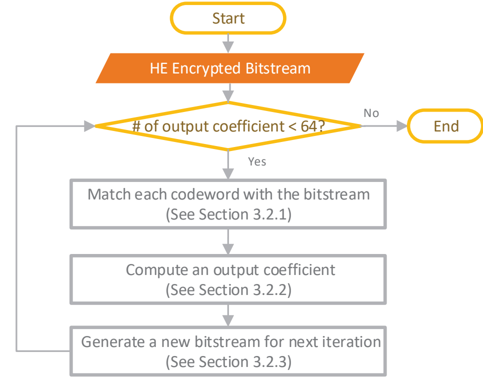


Figure 2: A flow chart of the proposed scheme to decode an 8×8 block.

ALGORITHM 1: Match a codeword

Input: Bits: $a_0, a_1, \ldots, a_{n-1}$, and encrypted bits: $[bits]_0, [bits]_1, \ldots, [bits]_{n-1}$.
Output: Encrypted $[b]$.

```
[b] = [1];
for i = 0 to n − 1 do
    if a_i = 1;
     then
        [b]=[b] * [bits]_i;
    end
    [b]=[b] * ([bits]_i + [1]);
end
```

3.2.1 *Determining A Matched Codeword.*

For a given encrypted bitstream, we need to determine which codeword in a Huffman table matches. As we have mentioned, Huffman tables are not encrypted. For an n-bit codeword, $a_0, a_1, \ldots, a_{n-1}$, Algorithm 1 compares with the first n bits of the encrypted bitstream, $[bits]_0, [bits]_1, \ldots, [bits]_{n-1}$, and outputs an encrypted bit, $[b]$, with $b = 1$ if the codeword matches the bitstream, and otherwise $b = 0$.

In the above algorithm, $[x] + [1]$ is equivalent to [NOT x] for a bit x. Huffman coding guarantees that there is one and only one codeword would match the bitstream at each iteration. That implies that, when Algorithm 1 is applied to all codewords in the current Huffman table, there is one and only one codeword with $b = 1$. All the other codewords would have their $b = 0$. This property enables bitwise computation for a computation on a sequence of bits, which will be used in Sections 3.2.2 and 3.2.3.

3.2.2 *Output One Coefficient.*

A codeword in a Huffman table may generate a zero-valued coefficient or a nonzero coefficient in an iteration. In the latter case, a number of bits determined by

ALGORITHM 2: VLI decoding to find an encrypted coefficient

Input: The number of bits n to read in and the number of bits m ($m \geq n+1$) to represent a coefficient, and encrypted bits: $[bits]_0, [bits]_1, \ldots, [bits]_{n-1}$.
Output: Encrypted coefficient $[C] \equiv [C_0][C_1] \ldots [C_{m-1}]$.
$[Sign] = [1] + [bits]_0$; // sign bit: $[Sign]$ = NOT $[bits]_0$
for $i = 0$ *to* $m - n - 1$ **do**
  $[C_i] = [Sign]$; // assign the sign bit and extend it.
end
for $i = m - n$ *to* $m - 1$ **do**
  $[C_i] = [bits]_{i-m+n}$;
end
$[C] = [C] + [Sign]$; // add bit $[Sign]$ to coefficient $[C] \equiv [C_0] \ldots [C_{m-1}]$.

Size in symbol S_1 associated with the codeword is read and VLI-decoded. This VLI-decoding process is executed as described in Algorithm 2: for $n \equiv Size$ bits, it outputs an encrypted coefficient $[C] \equiv [C_0][C_1] \ldots [C_{m-1}]$ represented in m bits, where $m \geq n+1$.

For each codeword, such as i-th codeword w_i, in the Huffman table, we calculate its boolean $[b^{w_i}]$ using Algorithm 1 and its output coefficient $[C(w_i)] \equiv [C_0^{w_i}][C_1^{w_i}] \ldots [C_{m-1}^{w_i}]$ if matched. Algorithm 2 is used if the coefficient is a nonzero coefficient. The output coefficient $[C] \equiv [C_0][C_1] \ldots [C_{m-1}]$ of the current iteration is just the sum of these output coefficients multiplied by their corresponding boolean:

$$[C] = \sum_i [C(w_i)] * [b^{w_i}].$$

By exploiting the fact that there is only one $b^{w_i} = 1$ for all the codewords $w_i \in$ the Huffman table, the above summation can be executed bitwise. For example, j-th encrypted bit $[C_j]$ of the output coefficient $[C]$ can be computed as:

$$[C_j] = \sum_i [C_j^{w_i}] * [b^{w_i}].$$

3.2.3 Generate a New Bitstream. Each codeword with more than one coefficient is associated with a replacement codeword that would replace it in generating a new bitstream. Codewords with symbols $(0, k)$ don't associate with any replacement. For EOB symbol $(0, 0)$, the codeword is kept without dropping until the last (i.e., 63rd) coefficient of the current block is output. In this way, each codeword w_i, if matched, would generate a new bitstream $[NewBits(w_i)] \equiv [NB_0^{w_i}][NB_1^{w_i}] \ldots$ by dropping its consumed bits, and then inserting its replacement codeword at the beginning should such a replacement codeword exist. The new bitstream $[NewBits] \equiv [NB_0][NB_1] \ldots$ generated at the end of current iteration is the summation of each codeword's new bitstream multiplied by the corresponding boolean that determines if the codeword is a match or not:

$$[NewBits] = \sum_i [NewBits(w_i)] * [b^{w_i}].$$

Like in Section 3.2.2, the above summation can be carried out bitwise. For example, j-th encrypted bit is calculated as:

$$[NB_j] = \sum_i [NB_j^{w_i}] * [b^{w_i}]$$

where $[b^{w_i}]$ carries the same meaning as in Section 3.2.2.

3.3 Adding Codewords

To enable our regularized iterative scheme to decode one coefficient per iteration, additional codewords may need to be added to Huffman tables. In general, if $S_1 = (15, 0)$ is used in a bitstream, then the following symbols must be added to the Huffman table: $S_1 = (Z, s)$ meaning s consecutive zeros, where $s \in [1, 15]$. If $S_1 = (r, s)$, $s > 0$ is used in a bitstream, then either symbol $S_1 = (r-1, s)$ or both symbols $S_1 = (Z, r-1)$ and $S_1 = (0, s)$ should be in the Huffman table. If a needed symbol is not in a Huffman table, it is added to the table.

Since all bitstreams are encrypted, our decoder does not know if a codeword in a Huffman table has actually been used by any bitstream or not. The worst scenario has to be assumed: any symbol in a Huffman table might be used. As a result, finding missing symbols can be readily done by examining all symbols in a Huffman table. For example, if the baseline Huffman tables are used, the only symbols we have to add to an AC Huffman table are just (Z, s) with $s \in [1, 15]$.

Symbols can be added into a Huffman table in several ways. We can check if the Huffman table has any room to take additional symbols. If it can take only a partial set of additional symbols, we can add a special *Escape* symbol to escape to an indexing table which contains the remaining symbols that cannot be added to the Huffman table. In this case, *Escape* combined with an index to an entry in the indexing table serves as a codeword in performing operations described in Section 3.2. In a rare case that no symbol can be added to a Huffman table, we would resort to the encoder to add symbol *Escape* to the Huffman table, which might have a tiny impact on JPEG coding efficiency. Otherwise added symbols have no impact on other JPEG operations or usage.

3.4 An Huffman Decoding Example

We will show the regularized Huffman decoding process for the 63 AC coefficients given in the example shown in Fig.1. We assume that the codewords for $(3, 5)$, $(2, 5)$, $(1, 5)$, and $(0, 5)$ are 11111110, 11111100, 11111000, 11100000, respectively, after adding codewords, if necessary, in the Huffman table. For simplicity, we assume that a coefficient is represented with 8 bits (using two's complement).

In the first iteration, only codeword 1111000, i.e. $(0, 6)$, has its boolean matching bit $b^{(0,6)} = 1$. Thus only its coefficient and newbits would be output in this round. Since its $Size = 6$, by reading the next 6 bits and applying the VLI-decoding described in Algorithm 2, the coefficient associated with the codeword is $[00111001] \equiv [57]$, which is the output of the current round. This codeword drop the consumed bits 1111000 111001 from the bitstream without inserting any codeword. Thus it produces a new bitstream [1111000 101101 1111111110011000 10111 1010 ...], which is the input bitstream of the next iteration.

In the second iteration, only codeword 1111000, i.e. $(0, 6)$, matches, with its $b^{(0,6)} = 1$. All other codewords' boolean bits are 0. Like the first iteration, this round would output coefficient $[45] \equiv [00101101]$ and a new bitstream [1111111110011000 10111 1010 ...].

In the third iteration, only codeword 1111111110011000, i.e., $(4, 5)$, matches. It will output a zero-valued coefficient $[0]$ and a new bitstream [11111110 10111 1010 ...] by dropping the consumed 1111111110011000 and inserting 11111110, the codeword

for $(3, 5)$. The next three iterations are similar. Each iteration outputs a [0] coefficient, and a bitstream of [11111100 10111 1010 ...], [11111000 10111 1010 ...], [11100000 10111 1010 ...], respectively.

In the 7th iteration, only codeword 11100000, i.e., $(0, 5)$, matches. By reading the next 5 bits and applying the VLI-decoding, its coefficient is $[23] \equiv [00010111]$, which is the output coefficient in this round, and its new bitstream is [1010 ...].

In the 8th iteration, only codeword 1010, i.e., EOF $(0, 0)$, matches. It will output coefficient [0] and the same bitstream for the next iteration. This will repeat until the 63rd iteration, in which [0] is output as the coefficient, and 1010 is dropped from the bitstream as the input bitstream for the next block.

4 IDCT AND OTHER OPERATIONS

Huffman decoding produces 64 quantized DCT coefficients per 8×8 block. Each quantized DCT coefficient is then dequantized by multiplying the corresponding element in the quantization table, which is an integer in $[1, 255]$ for 8-bit images. As mentioned in Section 2.1, n-bit integer multiplications can be performed on an FHT computer. For an unencrypted quantization table like in our encryption scheme, dequantization can be simply realized by left shifting and additions of n-bit integers. For example, $[C] \times 3 = ([C] << 1) + [C]$.

After dequantization, the DCT coefficients in a block are inverse DCT transformed. The inverse DCT transform in JPEG specification [24] is a float DCT transform that cannot be executed on a FHE computer. We adopt HEVC's integer inverse DCT transform [8] in our scheme. It applies 1-D integer inverse DCT twice, once on each direction. At the end of each 1-D DCT transform, a scaling factor is applied. Our scheme chooses scaling factor $S_{IT1} = 2^{-6}$ for the first 1-D inverse DCT and $S_{IT2} = 2^{-9}$ for the second 1-D inverse DCT. They correspond to right shifts by 6 and 9 bits, respectively.

4.1 Bit Depth of Integers in Decoding

For n-bit integer operations, the larger the value of n, the more complex the operation. We should minimize the bit depth of integers, i.e., the number of bits needed to represent integers, at each stage of JPEG decompression as much as possible. The bit depth at each JPEG decoding stage is discussed in this subsection.

4.1.1 Bit Depth of DCT Coefficients. Before applying forward DCT, pixel values of an input image are shifted from unsigned integer in range $[0, 2^8 - 1]$ to a signed integer in range $[-2^7, 2^7$fi?!1]. After forward DCT, the rounded DCT coefficients are integers in range $[-2^1 0, 2^1 0 - 1]$ [25]. Thus we need to use 11 bits to represent AC coefficients and 12 bits to represent differential DC coefficients (including the sign bit). Since the rounding operation in quantization can introduce errors, which may increase bit depth by 1 after dequantization. This means that bit depth $L = 12$ is sufficient to represent quantized and dequantized DCT coefficients as well as different DC coefficients.

4.1.2 Bit Depth of Integers in Inverse DCT. We adopt the worst scenario analysis used in Section 6.2.5 in [8] find out bit depth in each stage of the 1-D inverse DCT. Let the values to forward DCT are all -2^7, the largest value in magnitude. The output of the forward DCT is that only the DC coefficient is not zero. Inversely, assume that the DC coefficient is -2^{L-1}, the largest value in magnitude that can be represented by L bits. The output of the first 1-D inverse DCT is a matrix with the the first column elements are all equal to $-2^{L-1} \times 64 \times 2^{-6} = -2^{L-1}$ and other elements are 0, where 64 equals to the values of the elements in the first row of the 1-D inverse DCT. Thus the output of the first 1-D inverse DCT is L bit, and intermediate integers during the transform is $L + 6$ bits (to represent the worst integer $-2^{L-1} \times 64 = -2^{L+5}$).

By applying the second 1-D IDCT is applied to the output of the first 1-D IDCT, we obtain a matrix with all elements equal to $-2^{L-1} \times 64 \times 2^{-9} = -2^{L-4}$. Thus the output of the second 1-D IDCT can be represented with $L - 3$ bits, and intermediate integers during the transform can be represented with $L + 6$.

By using $L = 12$ from Section 4.1.1, we conclude that the bit depth is 12 bits for the output of the first 1-D IDCT and 9 bits for the output of the second 1-D IDCT, and the bit depth of intermediate integers in both stages is 18 bits.

4.1.3 Bit Depth in Removing Bias. The output values after the inverse DCT are represented in 9 bits, i.e., in the range $[-2^8, 2^8 - 1]$. By removing bias 128 introduced before the forward DCT at the encoding side, the resulting pixel values are in the range $[-2^8, 2^8 - 1] + 128 = [-128, 383]$, which requires 10 bits to represent. The final pixel values are in range $[0, 255]$, i.e., represented as an 8-bit unsigned integer.

4.2 Clipping Out-of-Range Pixel Values

Can we obtain the final pixel value just by dropping the first two bits after removing bias 128? The answer is no. This is because some resulting pixel values might stay outside the valid range due to quantization errors and finite accuracy in calculations. We need to clip out-of-range pixel values by setting negative values to 0 and values larger than 255 to 255. This is achieved with the following clipping scheme.

The input to the clipping scheme is a signed integer of 10 bits, denoted as $V_{in} \equiv b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$, while the output is an 8-bit unsigned integer, denoted as U_{out}. Let $U_8 \equiv b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ be the unsigned integer represented by the lowest 8 bits of V_{in}. If V_{in} is in the valid range, then $b_9 b_8 = 00$ and $U_{out} = U_8$. If $b_9 = 1$, then $V_{in} < 0$ and $U_{out} = 0$, This can be realized with $U_{out} = (1 - b_9) \times U_8$, which is 0 if $b_9 = 1$ and U_8 is $B - 9 = 0$. For a positive out-of-range value, $b_9 = 0$ and $b_8 = 1$, and $U_{out} = 255$. This can be achieved with $(1 - b_8) \times U_8 + b_8 \times 255$, which results in 255 if $b_8 = 1$ and U_8 if $b_8 = 0$. By combining these cases, we have the following regularized clipping equation on ciphertexts:

$$[U_{out}] = ([1] - [b_9]) \times (([1] - [b_8]) \times [U_8] + [b_8] \times [255]).$$

5 SIMD-PACKING AND ENCRYPTION

With our Huffman decoding described in Section 3.2, each block is decoded with the same sequential operations. This is also true for the remaining JPEG decompression operations described in Section 4. This property enables us to apply SIMD for parallelized Huffman decoding and calculations that decoding or calculating one block in the encryption domain corresponds to decoding or calculating D plaintext blocks, where D is determined by the parameters of homomorphic encryption. This desirable SIMD property

would also reduce the total size of ciphertexts since D plaintexts would be encrypted into one ciphertext.

We encrypt only entropy coded bitstreams, i.e., entropy-coded segments (ECSes), in a JPEG file. Using SIMD, D plaintext ECSes are encrypted into a single ciphertext ECS. D is typically in hundreds or thousands, such as $D = 256$ in our experiments. This can be implemented together with JPEG encoder. We choose the first D ECSes, with stuffing bytes removed if added. For each i-th bit in these ECSes, we form a vector of D dimension and encrypt it into i-th ciphertext. If bits in an ECS is exhausted, 0 is padded. This results in an encrypted ECS of N ciphertexts, where N is the number of bits of the longest ECS among the chosen D ECSes. N is treated as meta data of the ciphertext ECS that will be used by our JPEG decoder. This process is repeated until all the ECSes in the JPEG file are encrypted.

Our encryption scheme leaves the quantization table and Huffman tables unencrypted. This is because JPEG is typically used with the default value for the qualify factor and with the baseline Huffman tables encoding an image. This this case, both the quantization table and Huffman tables are known, and encryption of either table would facilitate known-plaintext attacks that adversaries have access to a set of pairs of plaintexts and corresponding ciphertexts. Another reason is that encryption of these tables would produce more ciphertexts than encryption of ECSes for the test images reported in Section 6.

How secure is our encryptions scheme? It is more secure than selective encryption popular for image and multimedia encryption [27]. Since the length of each ciphertext ECS equals to the longest length of the corresponding D plaintext ECSes, there are typically many possible combinations of the codewords in a Huffman table to match the longest length of the plaintext ECS. In addition, adversaries knows neither which plaintext ECS is the longest among the D plaintext ECSes nor the length of any ECS that is not the longest. We believe that our encryption scheme is practically secure.

If we choose to encrypt the quantization table, the proposed JPEG decoding is readily applicable: only the dequantization process needs to be modified to multiplications of two encrypted integers, which a FHE computer supports. If we choose to encrypt Huffman tables, the encryption side needs to add codewords to a Huffman table if needed, associate each Huffman codeword with its replacement codeword if there is any, encrypts each codeword as well as its $Size$ if the codeword is not associated with any replacement codeword. At the decoding side, Algorithm 1 needs to the modified to $[b] = \prod_i([a_i] + [bits]_i + [1])$, and we need to add a checking which one matches the encrypted $[Size]$ among possible values of $Size$ in Algorithm 2. This checking is similar to checking which codeword in a Huffman table matches the current encrypted bitstream described in Section 3.2.1.

6 IMPLEMENTATION AND EXPERIMENTAL RESULTS

6.1 Implementation and Experimental Setting

We have implemented the proposed scheme in C++ based on HElib [18], which implements BGV and supports bootstrapping, and the Independent JPEG Group's implementation of JPEG [17]. In HELib, a level is associated with a ciphertext, which cannot go beyond the lowest level. Addition of two ciphertexts retains the lower level of the two ciphertexts, while multiplication results in a level that reduces the lower level of the two ciphertexts by 1. When the lowest level is reached, no more computation can be performed on the ciphertext unless it is bootstrapped.

Our JPEG decoding uses additions and multiplications of a polynomial with a bounded degree on Z_2. It can be executed with LHE without any bootstrapping. In Algorithm 1, the matching boolean b of the longest codeword has the most reduction in the level, and the output new bitstream of the current iteration is determined by this b. This worst scenario prevailing makes ciphertext level reduce very fast, and the initial level has to be set high. We decided to bootstrap match booleans in our experiments to have a low initial level. For the experimental results reported in this paper, the following parameters were used for BGV in our experiments, $m = 4369$, $p = 2$, $d = 16$, $r = 1$, and the initial level $L = 41$, which led to $D = 256$, i.e., 256 plaintext ECSes were packed together and encrypted into one ciphertext ECS. In addition, we packed up to $d = 16$ ciphertexts in each ciphertext bootstrapping.

The experiments were carried out on a server with Intel Xeon CPU E5-2680 v4 of 2.40GHz with 14 physical (28 logical) Cores and 256GB of RAM running CentOS Linux release 6.5. Test images were all 8-bit 256×256 grayscale images. They were JPEG-encoded with the default value of 75 for the quality factor, and with one 8×8 block in an ECS. With the setting, there were 4 encrypted ECSes for an encrypted JPEG image.

6.2 Experimental Results

We collected the running time for JPEG decoding each test image. After JPEG decoding, an resulting image is still encrypted. We then decrypted it and compared with the original image to calculate the PSNR value. We ran experiments for both the baseline Huffman tables and the optimized Huffman tables were used in JPEG encoding. In the latter case, only the used codeword were contained in a Huffman table. For each case of Huffman tables, we conducted two sets of experiments. In the first set of experiments, only the first 30 DCT coefficients per 8×8 block were decoded. In the second set, all the 64 DCT coefficients in each 8×8 block were decoded.

Table 1 shows the running time with both cases of using the optimized Huffman tables and the baseline Huffman tables for 6 representative standard test images when the first 30 DCT coefficients of per 8×8 block were decoded. It also shows the corresponding PSNR of the resulting image (after decryption) in comparing with the original image. Table 2 shows the same results when all the 64 DCT coefficients per 8×8 block were decoded.

Fig. 3 shows the decoded image after decryption against the original image for the 6 test images shown in Tables 1 and 2 when both 30 coefficients and all the 64 coefficients per 8×8 block were decoded. We note that, for the same number of decoded DCT coefficient, the resulting image using the baseline Huffman tables is identical to the one when the optimized Huffman tables were used in encoding.

To verify the correctness of our JPEG decoder, we compared our decoded images, after decryption, with the decoded images without encryption using the JPEG decoder from the Independent JPEG Group [17], with the same number of DCT coefficients decoded.

They agreed within a rounding error of 1. When the HEVC's integer inverse DCT was used to replace the inverse DCT in the JPEG decoder of the Independent JPEG Group, they were identical. These experimental results have proved the correctness of our proposed JPEG decoding scheme on homomorphically encrypted JPEG as well as its implementation.

Table 1: Running time of decoding the first 30 DCT coefficients per 8×8 block and the resulting PSNR after decryption in comparing with the original image. Images are all 8-bit 256×256 grayscale images.

Image	Running Time (s)		PSNR
	Optimized Huffman	Baseline Huffman	(dB)
Barbara	7175.81	10510.36	36.79
Mandrill	6387.62	10454.95	33.03
Jetplane	7310.20	10517.57	37.38
Lena	6774.39	10518.38	35.53
Pepper	6876.15	10372.58	38.96
Walkbridge	6969.91	10780.00	33.65

Table 2: Running time of decoding all the 64 DCT coefficients per 8×8 block and the resulting PSNR after decryption in comparing with the original image. Images are all 8-bit 256×256 grayscale images.

Image	Running Time (s)		PSNR
	Optimized Huffman	Baseline Huffman	(dB)
Barbara	14785.22	21812.91	38.64
Mandrill	13177.91	22117.85	34.99
Jetplane	15020.84	21540.60	38.82
Lena	14103.90	21692.29	39.51
Pepper	14249.01	21909.66	39.80
Walkbridge	14520.20	21845.99	34.97

6.3 Discussions

From Tables 1 and 2, we can see that running times for different images under the same setting don't differ much, even though some images are much easier to compress than others. This can be explained by the fact that JPEG decoding in the homomorphic encryption domain realizes its worst-case behavior due to inaccessibility to encrypted contents. We can also see that the difference in running time when the baseline Huffman tables were used is smaller than that when the optimized Huffman tables were used. This is because different Huffman tables were used for different images in the former case, while the same Huffman tables were used for all the images in the latter case.

Another observation from Tables 1 and 2 is that the decoding time of JPEG images encoded with the optimized Huffman tables is about 30% to 40% shorter than that of JPEG images encoded with the baseline Huffman tables. This is because that a baseline table has much more codewords than an optimized Huffman table. Many codewords in a baseline table may not be used during encoding, but they have to be treated as if there were used in JPEG decoding in the homomorphic encryption domain since a FHE computer cannot distinguish used codewords from unused codewords.

Figure 3: Original (left), decoded images (after decryption) with the first 30 coefficients (middle) and all the 64 coefficients (right) per 8×8 block. The 6 images are in the same order as in Table 1.

Like other algorithms running in the homomorphic encryption domain, JPEG decoding in the homomorphic encryption domain is much (about 10,000 times) slower than the normal JPEG decoding. It is still impractical for most applications. A recent paper proposed

a faster FHE scheme that can perform bootstrapping in less than 0.1s [11], much faster than the HELib's bootstrapping, which takes about 75s on our test computer with same BGV parameters used in our experiments. This may bring hope of JPEG decoding in the homomorphic encryption domain in some real applications. We plan to use it and test its decoding time once its implementation supports SIMD.

7 CONCLUSION

In this paper, we presented a novel scheme that enables JPEG decompression in the encryption domain of leveled or fully homomorphic encryption. The scheme applies a statically controlled iterative procedure to decode one DCT coefficient per iteration. We also presented IDCT and other decompression operations in the encryption domain. Our scheme supports fully parallelized SIMD, which encrypts multiple plaintexts into a single ciphertext and processes one ciphertext corresponding to processing simultaneously multiple plaintexts. Our experimental results have proved the feasibility and correctness of our proposed scheme. Although a stride forwards, its execution is still too slow for most applications, a common drawback for executing algorithms in the homomorphic encryption domain. With advances of homomorphic encryption studies, it may eventually become a reality in practical applications. We hope that our scheme would shed a light on enabling more practical algorithms in the encryption domain.

REFERENCES

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2017. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *arXiv:1704.03578 [cs]* (April 2017). arXiv: 1704.03578.

[2] C. Aguilar-Melchor, S. Fau, C. Fontaine, G. Gogniat, and R. Sirdey. 2013. Recent Advances in Homomorphic Encryption: A Possible Future for Signal Processing in the Encrypted Domain. *IEEE Signal Processing Magazine* 30, 2 (March 2013), 108–117. https://doi.org/10.1109/MSP.2012.2230219

[3] Y. Bai, L. Zhuo, B. Cheng, and Y. F. Peng. 2014. Surf feature extraction in encrypted domain. In *2014 IEEE International Conference on Multimedia and Expo (ICME)*. 1–6. https://doi.org/10.1109/ICME.2014.6890170

[4] M. Barni, P. Failla, R. Lazzeretti, A. R. Sadeghi, and T. Schneider. 2011. Privacy-Preserving ECG Classification With Branching Programs and Neural Networks. *IEEE Transactions on Information Forensics and Security* 6, 2 (June 2011), 452–468. https://doi.org/10.1109/TIFS.2011.2108650

[5] T. Bianchi, A. Piva, and M. Barni. 2008. Implementing the discrete Fourier transform in the encrypted domain. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*. 1757–1760. https://doi.org/10.1109/ICASSP.2008.4517970

[6] Tiziano Bianchi, Alessandro Piva, and Mauro Barni. 2009. Encrypted Domain DCT Based on Homomorphic Cryptosystems. *EURASIP Journal on Information Security* 2009, 1 (Dec. 2009), 716357. https://doi.org/10.1155/2009/716357

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3 (2014), 13:1–13:36. https://doi.org/10.1145/2633600

[8] Madhukar Budagavi, Arild Fuldseth, and Gisle BjÃÿntegaard. 2014. HEVC Transform and Quantization. In *High Efficiency Video Coding (HEVC): Algorithms and Architectures*, Vivienne Sze, Madhukar Budagavi, and Gary J. Sullivan (Eds.). Springer, 141–169.

[9] Sebastien Canard, Sergiu Carpov, Donald Nokam Kuate, and Renaud Sirdey. 2017. Running compression algorithms in the encrypted domain: a case-study on the homomorphic execution of RLE. https://eprint.iacr.org/2017/392.pdf. (2017).

[10] Yu-Chi Chen, Chih-Wei Shiu, and Gwoboa Horng. 2014. Encrypted signal-based reversible data hiding with public key cryptosystem. *Journal of Visual Communication and Image Representation* 25, 5 (2014). https://doi.org/10.1016/j.jvcir.2014.04.003

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. 2016. Faster Fully Homomorphic Encryption : Bootstrapping in Less Than 0.1 Seconds. In *ASIACRYPT 2016*. 3 – 33.

[12] Zekeriya Erkin, Alessandro Piva, Stefan Katzenbeisser, R. L. Lagendijk, Jamshid Shokrollahi, Gregory Neven, and Mauro Barni. 2007. Protection and Retrieval of Encrypted Multimedia Content: When Cryptography Meets Signal Processing. *EURASIP J. Inf. Secur.* 2007 (2007), 17:1–17:20. https://doi.org/10.1155/2007/78943

[13] S. Fau, R. Sirdey, C. Fontaine, C. A. Melchor, and G. Gogniat. 2013. Towards Practical Program Execution over Fully Homomorphic Encryption Schemes. In *Int. Conf. on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. IEEE, 284–290.

[14] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*. ACM Press, 169–169. https://doi.org/10.1145/1536414.1536440

[15] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology – CRYPTO 2012*. Springer, Berlin, Heidelberg, 850–867. DOI: 10.1007/978-3-642-32009-5_49.

[16] Marta Gomez-Barrero, Julian Fierrez, Javier Galbally, Emanuele Maiorana, and Patrizio Campisi. 2016. Implementation of Fixed-Length Template Protection Based on Homomorphic Encryption With Application to Signature Biometrics. 191–198.

[17] Independent JPEG Group. 2016. JPEG Reference Code, version 9b. http://jpegclub.org/reference/reference-sources/. (2016). [Online; accessed Dec. 15, 2017].

[18] Shai Halevi. 2013. HElib: An Implementation of homomorphic encryption. https://github.com/shaih/HElib. (2013). [Online; accessed Dec. 15, 2017].

[19] R. L. Lagendijk, Z. Erkin, and M. Barni. 2013. Encrypted signal processing for privacy protection: Conveying the utility of homomorphic encryption and multiparty computation. *IEEE Signal Processing Magazine* 30, 1 (2013), 82–105. https://doi.org/10.1109/MSP.2012.2219653

[20] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. 2011. Can Homomorphic Encryption Be Practical?. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2046660.2046682

[21] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology — EUROCRYPT '99*. Springer, Berlin, Heidelberg, 223–238. https://doi.org/10.1007/3-540-48910-X_16

[22] Zhan Qin, Jingbo Yan, Kui Ren, Chang Wen Chen, and Cong Wang. 2016. SecSIFT: Secure Image SIFT Feature Extraction in Cloud Computing. *ACM Trans. Multimedia Comput. Commun. Appl.* 12, 4s (2016), 65:1–65:24. https://doi.org/10.1145/2978574

[23] N. P. Smart and F. Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography* 71, 1 (April 2014), 57–81. https://doi.org/10.1007/s10623-012-9720-4

[24] CCITT Recommendation T.81. 1992. Information Technology – Digital Compression and Coding of Continuous-tone Still Images – Requirements and Guidelines. (1992).

[25] Gregory K. Wallace. 1991. The JPEG Still Picture Compression Standard. *Commun. ACM* 34, 4 (April 1991), 30–44. https://doi.org/10.1145/103085.103089

[26] X. Zhang, J. Long, Z. Wang, and H. Cheng. 2016. Lossless and Reversible Data Hiding in Encrypted Images With Public-Key Cryptography. *IEEE Transactions on Circuits and Systems for Video Technology* 26, 9 (Sept. 2016), 1622–1631. https://doi.org/10.1109/TCSVT.2015.2433194

[27] Bin B. Zhu. 2006. Multimedia Encryption. In *Multimedia Security Technologies for Digital Rights Management*, Wenjun Zeng, Heather Yu, and Ching-Yung Lin (Eds.). Academic Press, 75–109.